Fachgebiet Programmierung eingebetteter Systeme
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin

A Framework for Machine Learning based Mapping of Concurrent Applications to Parallel Architectures

vorgelegt von
Diplom-Informatiker
Dirk Tetzlaff

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Ben Juurlink
Technische Universität Berlin

Berichtende: Prof. Dr. Sabine Glesner
Technische Universität Berlin

Berichtender: Prof. Dr. Peter Marwedel
Technische Universität Dortmund

Berichtender: Prof. Dr. Jens Knoop
Technische Universität Wien

Tag der wissenschaftlichen Aussprache: 24. Juni 2014

Berlin 2014
D 83

Abstract

In this thesis, we present a general framework to improve the automatic mapping of concurrent applications to parallel architectures and we define its instantiation for mapping MPI programs to processor networks. For scheduling the parallelly executable tasks of applications and for their allocation to the processing elements of the target architecture, the major challenge is to analyze in advance the expected run-time behavior of applications. Our proposed framework solves this problem by utilizing *Machine Learning* (*ML*) techniques to derive precise predictions for this information. This knowledge is used as fast and accurate heuristics to establish a cost model that rates the gain of various mappings. Using cost models based on machine learned heuristics that include knowledge about the run-time behavior of programs one can expect an advantage over cost models based on purely static analyses, which must conservatively over-approximate the run-time behavior. As a result, optimization potential to improve program performance is increased.

To improve the mapping, we define automatic analyses that statically determine the parallel structure of applications. Based on this, we introduce ML techniques to derive the most needed information for optimizing the mapping: knowledge about the execution times of parallelly executable tasks of applications and about the communication amount between tasks. These ML techniques have to be deployed only once per architecture in a training phase, which is decoupled from the compilations of applications. Hence, the compile time is not increased, thereby preserving an efficient and continuous compilation flow. To rate the gain of alternative mapping schemes, we also define a general cost model that is based on machine learned knowledge and that is parametrized by hardware-dependent information. Using this cost model enables a power-efficient and communication-aware mapping of applications to any parallel architectures.

Our general framework is applicable to a wide diversity of parallel programming models and target architectures. In the second part of this thesis, we show how our general framework can be applied for improving the mapping of MPI programs to processor networks. We have fully implemented the instantiated framework and performed experiments to determine the accuracy of our approach. For our experiments, we have used a considerable number of programs from various benchmark suites that encompass different real-world application domains. This shows on the one hand the general applicability and on the other hand the high scalability of our framework. The evaluation of our experiments demonstrates that we are able to predict regarded run-time behavior more precisely compared to other heuristic approaches.

Zusammenfassung

In dieser Arbeit stellen wir ein generelles Framework zur Verbesserung der automatischen Abbildung von nebenläufigen Anwendungen auf parallele Architekturen vor. Die größte Herausforderung bei der notwendigen zeitlichen Planung der parallelen Tasks und bei deren Zuordnung zu Berechnungseinheiten der Zielarchitektur ist, vorab das erwartbare Laufzeitverhalten der Tasks zu bestimmen. Unser Framework löst dieses Problem durch Einsatz von Techniken des *Maschinellen Lernens* zur präzisen Vorhersage dieser Information. Dieses Wissen verwenden wir als schnelle und genaue Heuristik um ein Kostenmodell zu etablieren, welches die Qualität der Abbildung bewertet.

Vorhersagen mit Hilfe von Techniken Maschinellen Lernens, die Informationen über das Laufzeitverhalten zur Verfügung stellen, lassen präzisere Ergebnisse erwarten als Vorhersagen, die auf rein statischen Analysen beruhen, welche das Laufzeitverhalten approximieren müssen. Dadurch erhöht sich das Optimierungspotential zur Verbesserung der Leistungsfähigkeit von Programmen zur Laufzeit.

Das in dieser Arbeit entwickelte Framework besteht aus zwei voneinander getrennten Phasen: eine einmalige Trainingsphase pro Architektur vor der Anwendungsphase des Frameworks. Während der Trainingsphase werden mittels Maschinellem Lernen Verhaltensmodelle erzeugt, die einen Zusammenhang zwischen statischen Eigenschaften der Programme und deren Laufzeitverhalten herstellen. Während der Anwendungsphase des Frameworks werden diese dann verwendet, um den Berechnungsaufwand von Tasks und Kommunikationskosten zwischen Tasks vorherzusagen.

Das entwickelte Framework ist für diverse parallele Programmiermodelle anwendbar. Im zweiten Teil der Dissertation stellen wir die Instanziierung des Framework für das *Message Passing* Programmiermodell vor. Wir haben dieses instanziierte Framework vollständig implementiert und Experimente durchgeführt, um die Qualität der Verhaltensvorhersagen zu ermitteln. Die Evaluierung der Experimente zeigt, dass wir nicht nur das betrachtete Laufzeitverhalten besser vorhersagen können als andere heuristische Verfahren, sondern auch die Laufzeit von Programmen mithilfe unseres Frameworks deutlich verkürzt werden kann.

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter im Fachgebiet “Programmierung eingebetteter Systeme” unter der Leitung von Prof. Dr. Sabine Glesner. Bei ihr möchte ich mich an dieser Stelle herzlich für die Betreuung meiner Arbeit bedanken. Sie hat mir den Freiraum gelassen, eigene Schwerpunkte zu setzen, und ohne ihre Bereitschaft zu Diskussionen und ihre hilfreichen Kommentare wäre die Arbeit so nicht zustande gekommen. Herrn Prof. Dr. Marwedel und Herrn Prof. Dr. Knoop gilt mein Dank für die Durchsicht und Begutachtung meiner Arbeit.

Neben den Gutachtern möchte ich mich bei zwei Personen ganz besonders bedanken: Dr. Paula Herber und Dr. Thomas Göthel. Sie haben nicht nur während meiner gesamten Promotionszeit durch ihre wissenschaftlichen Anmerkungen zum Gelingen dieser Arbeit beigetragen, sondern mich auch bei meinem Endspurt bis zum heutigen Tag durch wiederholtes Korrekturlesen und hilfreiches Feedback unterstützt. Weiterhin möchte ich Dr. Lars Alvincz sehr danken für seine Anregung zu einem Schwerpunkt meiner Arbeit und seine Hilfe bei der Einarbeitung in dieses Thema, sowie Dr. Michael Beyer für die freundliche Arbeitsatmosphäre in unserem Büro. Auch allen anderen Kolleginnen und Kollegen in unserem Fachgebiet danke ich für die angenehme Atmosphäre in unserem Fachgebiet. Ich werde die Frühstücks- und Raucherpausen vermissen.

Zuletzt möchte ich meiner Familie und meinen Freunden danken für ihre Unterstützung während der ganzen Zeit. Insbesondere gilt mein größter Dank meiner Freundin Liane Herzmann und meinem Freund Heino Gierke, welche mir immer den Rücken freigehalten haben und mich privat auch mit ihrer guten Laune unterstützt haben, wo sie nur konnten. Einen ganz lieben Dank vor allem auch an meine Mutter Renate Tetzlaff, die mich nicht erst während meines Studiums und meiner Promotionszeit unterstützt hat, sondern ein Leben lang, und meinem Bruder Andreas Tetzlaff, der mir in letzter Zeit einige familiäre Verplichtungen abgenommen hat.

Dirk Tetzlaff, Januar 2014

Contents

1 Introduction

As a matter of fact, the steadily increasing demand for higher performance rules all application domains. Parallelism is one of the main ways to improve performance, but the efficient exploitation of available parallelism depends on a number of factors. Of main importance is a mapping scheme that improves the run-time performance of a concurrent application by optimally allocating its parallelly executable tasks to the *Processing Elements* (*PEs*) of the target architecture. The major challenge of deriving an optimal mapping is to obtain in advance the needed information about expectable run-time behavior of these tasks. Because mapping should be done automatically, this has to be realized by automatic analyses as used in compilers.

1.1 Problem

The problem of optimal task mapping to a limited number of PEs is a computationally complex challenge since the corresponding decision problem for partitioning is NP-complete [Kar72]. This requires the use of heuristics to find near-optimal solutions for real-world applications in an acceptable timeframe. For improving task mapping, these heuristics demand the run-time behavior of tasks to be known. To facilitate an automatic mapping, this information has to be provided by *automatic* analyses. Furthermore, improving an initial mapping or improving static mapping techniques requires the analyses to *statically* predict the run-time behavior in advance. This is a major challenge because solely analyses at run-time are able to predict the behavior of periodic tasks reasonably accurate, and even then not until the first execution of these tasks. Moreover, analyses that infer predictions from information collected at run-time can either exclusively be used to direct heuristics at run-time or, to be used in a static compiler, their result must be fed back to an additionally required compilation step after the execution of the application that delivers the run-time behavior. For the former case, e. g., heuristics for a dynamic run-time scheduler, an initial mapping or static task mapping techniques are not improved since run-time information is first gathered after the compilation. The latter case, called *feedback directed optimization* (*FDO*), requires instrumentation of code during an initial compilation step to specify

run-time information that has to be collected. Then, instrumented code must be executed sufficiently many times with typical input data for each execution until all information is collected, which is called *profiling*. Additionally to the executed code of the application, profiling executes operations for analyzing and storing necessary run-time values to derive needed information, which may include complex calculations with a substantial execution time overhead. Gathered run-time behavior can then be exploited by heuristics during a second compilation step. While being effective in improving run-time performance of applications, instrumentation-based FDO has not been widely adopted due to the tedious dual-compilation model, the difficulties in generating representative input data sets, and the high run-time overhead of profile collection [CVH+13]. To mitigate the overhead, profile collection can be limited to an one-off execution run, though having the drawback to gather only a single run-time behavior, not expected realistic behavior.

In conclusion, the needed run-time behavior cannot be known beforehand in general – not at run-time and even less statically at compile time without profiling, which is usually ruled out in terms of prohibitive overhead for profiling and dual compilation. Therefore, the compiler has to use in this case heuristics based on purely static analyses, which conservatively over-approximate. This can have severe consequences on the run-time performance of compiled applications because the most important information for task mapping, the expected execution time of tasks and the communication amount between tasks at run-time, are hardly predictable using static techniques.

1.2 Objectives

The aim of this thesis is to establish a compiler framework for improving the mapping of concurrent applications to parallel architectures. As general objectives, we require the framework to provide automatic, efficient, and highly scalable methods for statically analyzing the expected run-time behavior of applications and, based on this, for optimizing the run-time performance of applications. To assess the quality of our framework, these objectives imply the following criteria that must be fulfilled:

- **Automatic Application of the Analyses** The analyses of the framework should be automatically applicable at compile time without the need for user intervention.
- **Precision of the Analyses** The analyses of the framework should yield precise results because heuristics that use their results rely on the outcome to determine the benefit of subsequent transformations.
- **Efficient Compilation** The framework should ensure an efficient compilation such that the compile time is not substantially increased. This requires the analyses to determine the analytical result in an acceptable timeframe, but without forfeiting the precision.

- **Continuous Compilation Flow** The framework should preserve a continuous compilation flow such that a user does not have to deal with interrupting the compilation due to the fact of being obliged to use external tools. This requires the analyses and applied transformations to be integrated into the compiler framework.
- **Scalability of the Heuristics** The heuristics of the framework should be highly scalable to be applicable within compiler environments used for compilation of complex real-world applications.
- **Generality of Parallel Programming Models** Since we aim at improving the run-time performance of any concurrent application, the framework should be general enough to be applicable to various parallel programming models.
- **Generality of Target Architectures** Since we aim at improving the mapping of concurrent applications to any parallel architecture, the framework should target architectures with unrestricted structure.
- **Generality of the Cost Model** Because it is necessary to establish a cost model that rates the expected gain of the mapping to different architectures, the cost model must be parameterized by hardware-dependent information.
- **Generality of Application Domains** The framework should be suitable for highly diverse application domains to be applicable to any real-world application.

1.3 Proposed Solution

To achieve the objectives defined above, in this thesis we present a general framework for automatically improving the mapping of concurrent applications to parallel architectures. We tackle the problem of incorporating statically unknown or imprecise needed information into the compilation flow with the use of *Machine Learning* (*ML*) techniques. To obtain the necessary training data for ML, we perform several profiling runs with different input for a comprehensive suite of programs during a one-off *training phase* per architecture prior to the actual compilation of applications. This data is condensed and abstracted with ML techniques into a model that relates static properties of an application to its dynamic behavior at run-time. The machine learned model is automatically transformed to heuristics that predict the regarded run-time behavior of applications solely based on static code features. Hence, it is only necessary to statically analyze a number of code features to obtain a prediction. These analyses are typically computationally less intensive than corresponding program analyses.

Though static analyses also can estimate run-time behavior, the estimates are imprecise because their analytical results must consider all possible cases how a program can behave, e. g., all different behaviors resulting from var-

ied input data. Hence, run-time behavior is rarely predictable using static techniques due to this over-approximation. Therefore, approaches to obtain this run-time information were proposed. However, profile-guided compilation or manual annotations of all necessary forecast values to include information about the run-time behavior have drawbacks. While both approaches are prohibitively time consuming, the former is additionally strongly dependent on the input data set of each profiling run because it works well only when the actual run-time program tendencies match those collected during profiling [Smi00], and the latter is additionally difficult and error-prone or often an impossible burden for the programmer since not all run-time information is known. In contrast, our approach eliminates the need for profiling and manual annotations during compilation, while automatically generating heuristics which are closely connected with the anticipated behavior. Moreover, the beforehand machine learned model holds observations from several profiling runs, which has the advantage to focus the mapping on more realistic behavior, not on one single behavior as it is the case for profile-guided compilation with only one execution run of instrumented code.

Our approach automatically generates architecture- and application-specific heuristics that efficiently provide the compiler with knowledge of run-time behavior. This bridges the gap between static program analyses on the one hand and dynamic run-time behavior on the other hand and makes it possible to improve the mapping of concurrent applications to parallel architectures. Since the heuristics are automatically generated and incorporated into the compiler without the need for user intervention, our approach preserves an automatic and continuous compilation flow. Note that the generated heuristics can also be combined with static program analyses. When the analysis knows the results to be exact, these results can be used, otherwise, the heuristics is consulted. Thereby, our approach combines the benefits of static analyses and profiling without taking their disadvantages. While the training phase entails a significant overhead, it is only executed once per architecture decoupled from the compilation, so the compile time for applications is not increased. Due to the high scalability of the resulting heuristics derived from the machine learned model, the additional compile-time overhead is negligible and lower than that of conservative program analyses. Thus, our approach also preserves an efficient compilation flow.

The most importantly needed run-time information for determining the best mapping scheme are the expected execution times of the parallel tasks of an application and the communication amount between tasks at run-time, whose overheads have emerged as the major performance limitation. In general, there is a trade-off between balancing the workload to minimize execution time and reducing communication cost. On the one hand, allocating tasks that communicate with each other to the same PE removes the associated communication cost, but results in poor workload balance. On the other hand, balancing the workload typically increases communication. Task mapping must therefore find an allocation scheme in which both criteria are weighted against each other. To that end, we propose to use a precise cost model that considers

corresponding run-time behavior of applications and that is parametrized with hardware-dependent information for determining the best mapping scheme.

Our general framework for ML based mapping of concurrent applications to parallel architectures is applicable to a wide diversity of parallel programming models because we do not constrain the programming languages. To show its practical applicability, we utilize our framework to improve the mapping of *Message Passing Interface* (*MPI*) programs to arbitrary processor networks. This also demonstrates the general applicability of our framework to any target architecture. To determine the quality of our framework, we have performed experiments for evaluating the precision of run-time behavior predictions. Using many programs of various benchmark suites from different real-world application domains for our experiments shows the general applicability of our approach to a wide range of programs with different behavior. In [TG10], we already sketched the idea of using *Machine Learning* to improve the mapping and presented first experimental results. In [TG12a], we showed how our framework can improve the mapping of MPI programs and presented experimental results that demonstrated the precision of our run-time behavior predictions.

1.4 Motivation

Task allocation is a vital part of mapping concurrent applications to parallel architectures. Finding an optimal solution requires information about execution times of tasks during run-time of applications, which is not known in advance at compile time. Furthermore, it must take into account interprocessor communication at run-time, whose overheads have emerged as the major performance limitation in concurrent applications owing to transmission delays, synchronization overheads, and conflicts for shared communication resources created by data exchange. Even with aggressive task partitioning and communication optimizations, there is still a lot of interprocessor data communication in a typical real-world application. Moreover, communication contributes to increased power consumption. Hence, the allocation of multiple interacting tasks of a concurrent program to the PEs of a parallel architecture must focus on communication costs. However, most interprocessor data-communication requirements can only be known at run-time and cannot be detected or optimized away by the compiler. Therefore, one has to predict loop iteration counts if interprocessor communication arises within loop bodies that do not have statically determinable loop bounds. Likewise, if interprocessor communication arises within recursive functions, one has to predict their recursion frequencies at run-time. But pure static analyses cannot predict iteration counts nor recursion frequencies precisely. Consequently, it is necessary to automatically incorporate knowledge of related dynamic behavior into the compiler but without the need for manual annotations or profiling to preserve an automatic, continuous and efficient compilation flow. Moreover, the technique must be highly scalable to facilitate the integration in industrial-strength compiler environments used for compilation of real-world applications.

In the case of heterogeneous multiprocessor systems, allocating the task to the PE that executes the code most efficiently adds another constraint to the allocation problem. Besides the ever-increasing demand for higher performance, recent research focuses strongly on energy-aware applications. Both objectives – maximizing performance and minimizing power consumption – are conflicting requirements because lowering supply voltage reduces power quadratically but also results in a performance degradation of the PE. Circuit-level power minimization techniques can be used to address the power concern, e. g., *Dynamic Voltage and Frequency Scaling* (*DVFS*) [Qu01].

With this thesis, we present a novel approach to automatically generate architecture- and application-specific heuristics for power- and communication-aware mapping of concurrent applications to parallel, heterogeneous architectures. To that end, we use ML techniques that yield more precise heuristics than those based on pure static analysis and that also scale for large applications. Note that our solution is not limited to static task mapping. Instead, the established heuristics can be used to direct the dynamic run-time scheduler, which in general cannot predict the execution time of tasks in advance without our approach. We aim at determining communication overhead via predicting loop iteration counts and recursion frequencies. Beyond that, both types of information can be used to guide further *profile-based optimizations* like the approaches of Hotta et al. [HSK+06] or Feng et al. [FDM+08]. Moreover, it enables, for example, *hot path* identification to guide various optimizations [BMYO07, DB00] because loops and recursive functions are the two kinds of programming structures where most of the execution time is spent.

1.5 Main Contributions

The main contributions of this thesis are:

General Framework for Task Mapping We propose a general framework for mapping parallelly executable tasks of applications to PEs of any parallel architecture. Furthermore, our framework is applicable to a wide diversity of parallel programming models because we do not constrain the programming languages.

Machine Learning for Precise Run-time Prediction We tackle the problem of incorporating information about the execution time and communication overhead during run-time of applications into the compilation flow using ML techniques to bridge the gap between static program analyses on the one hand and dynamic program behavior on the other. This facilitates better results than those generated by pure static analyses, since the analyses may focus on realistic behavior. In contrast, estimates based on overly conservative static assumptions must consider all possible behavior and therefore drastically over-

approximate, and one execution of instrumented code for profile collection considers only one single behavior, hence being too restricted.

General Cost Model Using cost models based on machine learned heuristics that include knowledge about the run-time behavior of programs one can expect an advantage over cost models based on purely static analyses, which must over-approximate the run-time behavior. As a result, optimization potential to improve program performance is increased. Our proposed general cost model is parametrized by hardware-dependent information and considers execution times of tasks and communication amount between tasks, which enables a power-efficient and communication-aware mapping of applications to any parallel architecture.

Intelligent Mapping of MPI Programs We present the application of our general framework to the optimization of mapping MPI programs to processor networks. Using our ML based knowledge to derive a cost model for the allocation of MPI processes to PEs makes the mapping more intelligent and thus improves the run-time performance.

Practical Evaluation To determine the accuracy of our approach, we first evaluate experimental results for learning needed run-time behavior in detail and compare them with results of other heuristic approaches. The evaluation of our experiments shows that, as is the case in many other domains, programs can be successfully represented by static code features. These features are then used to gauge their similarity and thus the applicability of previously learned off-line knowledge. This solves the problem of over-approximation, which is inherent to static program analyses. Second, we have performed experiments to investigate the performance gain achieved with our approach. Our experiments demonstrate that we are able to thightly predict the regarded run-time behavior and, based on this, the run-time performance of programs is significantly improved.

1.6 Overview of this Thesis

This thesis is structured as follows: In Chapter 2, we give necessary background information on compiler construction, on ML techniques, and on task mapping. Especially, we describe the role of program analyses during compilation. For ML, we introduce the concepts of classification learning and regression modeling, which are central for our approach, and we characterize quality measures of the ML model. For task mapping, we discuss parallel programming models and parallel target architectures. In Chapter 3, we review related work on using ML in compilers, on predicting run-time behavior of applications, and on task mapping. In Chapter 4, we present our general framework and we describe the entailed phases in detail. We also revisit our

criteria for the general framework and discuss whether or not they are met by our framework. Chapter 5 shows how our general framework is applied to improve the mapping of MPI programs to processor networks. Chapter 6 characterizes the implementation of our applied framework. In Chapter 7, we evaluate the benefit of our framework. We first present our large collection of benchmark suites that comprises real-world programs from diverse application domains. Then, we demonstrate the accuracy of our ML based run-time behavior predictions and we show the overall performance gain of programs achieved due to our framework. Chapter 8 summarizes the results of this thesis and gives an overview of future work.

2 Background

In this thesis, we present a compiler framework for improving the mapping of concurrent applications to parallel architectures as well as its instantiation for mapping *Message Passing Interface* (*MPI*) programs to processor networks. The central idea is to use *Machine Learning* (*ML*) techniques to precisely derive needed information about the expectable run-time behavior of applications. Thus, this thesis is embedded in two broader areas of research, compiler construction, which is a subfield of software engineering, and ML, which is a subfield of artificial intelligence. In the following we first provide the necessary background knowledge on compiler construction in Section 2.1. Vital to compiler optimizations are program analyses, which we review afterwards in Section 2.2. We continue in Section 2.3 with describing ML techniques, namely classification learning and regression modeling, and we characterize quality measures of the ML model. Since we aim at improving the mapping of arbitrary applications to any target architectures with this thesis, we next characterize parallel programming models in Section 2.4 and parallel architectures in Section 2.5. Finally, we describe the challenge of task mapping in Section 2.6.

2.1 Compilers

A compiler is a computer program that converts words of one formal language into other words of a formal language. We exclusively consider compilers that statically transform source code written in a high-level programming language into executable machine code for a given target platform. These compilers enable the development of applications that are architecture-independent. Thus, the developer has the freedom to abstract from architectural details and to focus on implementing a cleverly thought out solution of his issue. As a consequence, the compiler has now the obligation to generate performant machine code that best utilizes resources of the target platform. In order to optimize the code, compilers need indications on performance of language constructs regarding the target architecture. This is a major challenge because used compiler analyses for rating the optimization gain during compilation have to predict the performance at run-time in advance.

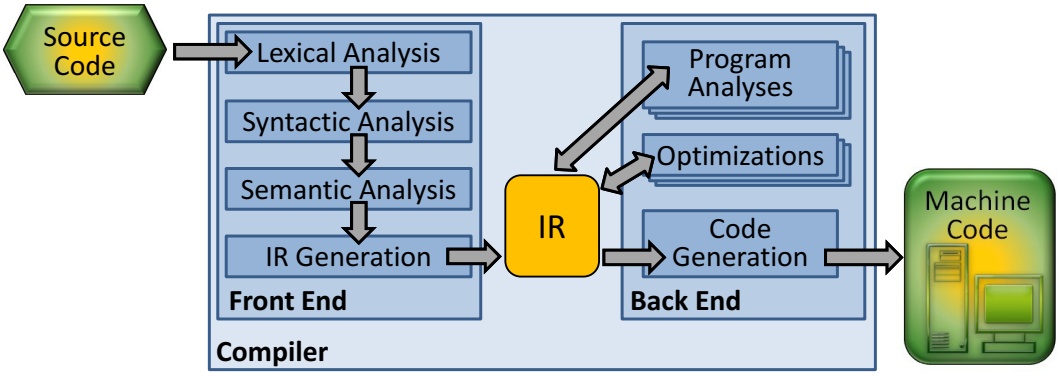


Figure 2.1: Compiler structure

Compilers typically first convert the source code to an *Intermediate Representation* (*IR*), which then is transformed by subsequent optimizations. Program analysis is the prerequisite for any compiler optimization, and they tightly work together. Finally, the IR is translated into machine code. Figure 2.1 depicts the structure of such a compiler. The following subsections describe the *front end* that generates the IR from source code, the role of the IR during compilation, and the *back end* where optimizations based on program analyses take place and machine code is produced.

2.1.1 Front End

The front end analyzes the source code to build the IR as an internal representation of the program. This procedure can be divided into four phases, lexical, syntactic, and semantic analysis, followed by the IR generation. The lexical analysis breaks the source code text into single atomic units of the language called *tokens*. The syntax analysis identifies the syntactic structure of the program. This involves parsing the token sequence to check whether identifiers are used appropriately within each statement or whether the source code contains syntactic errors such as that the number of opening brackets is unequal to the number of closing brackets. If no errors are present, the linear sequence of tokens is transformed to a tree structure, called *Abstract Syntax Tree* (*AST*), which is built according to the rules of a grammar that defines the syntax of the language. Semantic analysis is the phase in which the compiler adds semantic information to the AST and builds the symbol table that associates lexical names (symbols) in a program with their attributes[1]. This phase also performs semantic checks such as requiring all local variables to be initialized before being used or checking for type errors. In case of semantic consistency, the IR is generated in the last phase of the front end. This is done via recursive traversal of the AST, making control structures like loops or `if`-then-else statements explicit.

[1]Some compilers set up a table at lexical analysis time for the various variables in the program, and fill in information about the symbol later during semantic analysis when more information about the variable is known.

2.1.2 Intermediate Representation

The *Intermediate Representation* (*IR*) is a combination of different data structures representing the program under compilation. Common in compiler construction is the use of *multilevel* IRs, i. e., different IRs on different abstraction levels, each level supporting corresponding optimizations. These IRs are categorized according to where they fall between a high-level programming language and machine code, e. g., the *High-Level IR* (*HIR*) that has source language dependent constructs and the *Low-Level IR* (*LIR*) that has target language dependent constructs. The HIR is stepwise transformed to LIR, which is called *lowering*. HIRs typically preserve information such as array subscripts, the loop structure, or `if`-then-else statements, whereas in lower levels those are converted to explicit addresses or conditional `goto` statements, respectively. The LIR is suitable for machine-dependent transformations like *register allocation* and *instruction selection*. Usually, there is a third level in between the both mentioned above, the *Medium IR* (*MIR*) that is independent of both the source and the target language. However, some compilers encompass less levels, like the *CoSy*© compiler of ACE [ACE13] that does not define a HIR. Instead, the AST is regarded as the HIR and the IR starts at the medium level, called the *Common CoSy Medium-level Intermediate Representation* (*CCMIR*). Besides, some compilers encompass more levels, like the *Open64 Compiler* [Liu07]. The Open64 Compiler defines the AST as the *Very* High-Level IR and additionally introduces the *Very* Low-Level IR, where, e. g., symbolic registers are replaced by actually existing target registers.

The typical main data structure in the IR is the *Control Flow Graph* (*CFG*) that represents the structure of one single function as a directed graph. In most CFG representations, the nodes are *basic blocks* (*BBs*) that exclusively contain maximal *straight line code* (also called *branch-free code*) and edges represent jumps in the control flow. That is, solely code without any jumps or jump targets are within a BB except that jumps can end a block (corresponding to an outgoing edge of that node) and jump targets start a block. There are special BBs and edges in the CFG. The unique *entry BB* is the first block of the function, the unique *exit BB* is the block where the function exits. If the exit is not unique, e. g., due to several `exit` or `return` statements, a further BB is constructed that is the successor of all blocks where the calculation of the function terminates. Special edges are *back edges*, targeting a block that has already been met during a depth-first traversal of the graph. Back edges are typical for loop structures. Identifying loops, as can be done via *Natural Loop Analysis* [ALSU07] in reducible CFGs[2], is a prerequisite for several optimizations, such as *Loop Invariant Code Motion* or *Loop Unrolling*.

Besides the CFGs for each function, the IR holds data types and variables defined in the program as well as imported definitions. Additionally, a static *Call Graph* (*CG*) can be constructed as a basic program analysis result. A static CG is a multigraph that captures every function call relationship between a *caller* and the *callee*. Specifically, each node represents a function and

[2]The CFG is *reducible* if and only if for every back edge $v \to w$, either $v = w$ or $v \neq w$ and w is on every path from the entry BB to v.

each edge (f, g) indicates that function f calls function g. Thus, a cycle in the graph indicates recursive function calls. This CG facilitates interprocedural analyses, such as the approach of Wu and Larus [WL94] for static program profile analysis, and interprocedural optimizations like the approach of Ferraris [Fer11] for inline expansion of functions. Static program analyses, which are introduced in more detail in Section 2.2.1, commonly annotate their result to the IR and optimizations may alter the data structures of the IR, thereby using analytical results to rate the gain of a transformation.

2.1.3 Back End

In the back end, consecutive transformations of the IR are performed aiming at improving the representation w. r. t. an objective function, whereby the semantics of the program has to be preserved. These optimizations use results of program analyses to ensure program correctness and to decide which transformation will be applied. A special case of an optimization is *lowering*, which transforms the IR to lower levels where constructs of the IR are replaced by more target specific ones or are enriched with information about the target architecture. Lowering to LIR yields a representation which is very close to the assembler code.

In general, optimizations can be classified as *optimal* or *heuristic*. To attain their objective function, optimizations have to determine which transformation will be applied. If this search space of possible transformations is explored *exhaustively* to check all transformations against each other for the maximal gain w. r. t. the objective function, the optimization is called *optimal* because its result is guaranteed to be the best applicable transformation. One popular technique for determining the optimal transformation is to define the optimization problem and a linear objective function in a mathematical model where requirements are represented as linear relationships, which is called *Integer Linear Programming* (*ILP*). However, due to the state space explosion of possible transformations for more complex optimizations, it is not viable to explore the search space exhaustively in an acceptable time frame as it is required for compiler optimizations. In these cases, *heuristic* optimizations that explore the search space only *partially* can be used whenever solving a problem that ignores whether the solution can be proven to be optimal. Although heuristics do not guarantee that an optimal solution is ever found, this usually produces a good solution or solves a simpler problem that contains or intersects with the solution of the more complex problem.

The typical objective function of the overall optimization sequence is to minimize the execution time of the generated program or, less commonly, its resource usage, like the amount of memory occupied or the power consumed during execution. However, no compiler is able to guarantee that the best optimization sequence is applied for all program source code. Such a compiler is fundamentally impossible because it would solve the *Halting Problem*, which is undecidable. This can be shown by considering a call to a function `f`() that returns nothing and does not have side effects. The fastest possible transfor-

mation would be simply to eliminate the function call. But if this function in fact does not return, then the program with the call would be different from the program without the call. A compiler that guarantees to find the best optimization sequence would then have to determine this by solving the Halting Problem.

The final step in the back end is code generation which works on the LIR. Code generation is a particular phase that can be divided into the three optimization steps *instruction selection*, *instruction scheduling*, and *register allocation*, and the endmost step of emitting machine code. Some literature uses the term *middle end*, e. g., for the *gcc* compiler [Nov04], to distinguish the generic analysis and optimization phases in the back end from the code generation phase. Since analyses are crucial for all optimizations, we review program analyses in the next section.

2.2 Program Analyses

Program analysis is the process of automatically analyzing the behavior of computer programs. For compilers, program analyses are vital to identify optimization opportunities and to ensure that the program behavior is not changed. One possible classification of program analyses is whether they determine the result through pure *static analysis* of the IR or through executing the program under compilation with a typical input data set, which is called *profiling*. Profiling, which can be seen as a *dynamic* program analysis, considers only one execution for the given input data and hence only one possible run-time behavior. Therefore, its result is not safe, i. e., a transformation that uses this result cannot rely on the correctness of the outcome. Static analyses on the other hand can consider the entire expectable behavior and thus are required for transformations that demand safe analytical results. In the following we introduce both concepts, static program analysis and profiling.

2.2.1 Static Analyses

Static program analyses are performed without actually executing programs. They inspect the IR of the program under compilation and derive their outcome solely based on this static information. The scope of analyses varies from as small as only considering the properties of individual statements, via considering characteristics of BBs in the CFG or characteristics of function bodies, through to regarding the whole program behavior as it is the case for *interprocedural analyses*. Since these analyses are static, they can only make assumptions about how the program may behave at run-time. If the analytical result is required to be correct in all cases it must be a safe approximation of the expectable behavior. This is met by formal static analyses that use rigorous mathematical models, like *Data-flow Analyses* (*DFAs*), *Abstract Interpretation* (*AI*), or *Model Checking*. The obligation to consider every possible eventuality enforces safe static analyses to over-approximate, which can

lead to highly imprecise results. One such example is the *points-to-analysis* that produces a conservative approximation of the possible sets of variables, data structures, or functions a particular pointer could point to at a specific program point. The work of Mock et al. [MDCE01] shows that the statically determined points-to-set sizes range from 1 to 177 for the best of the scalable static pointer analysis algorithms, while the actual dynamic points-to set sizes were on average (geometric mean) a factor of 5 smaller. Additionally, for over 97% of dereferences the dynamic points-to sets were actually singletons. As a consequence, optimizations that use their analytical result are limited and optimization potential is sacrificed.

If analyses may yield incorrect results because transformations based on them do not affect the semantics of the program, they can neglect unlikely behavior. An unsafe analysis can therefore accomplish its task by *heuristically* estimating the expectable behavior, which has the potential of being more precise than safe analyses. Hence, heuristic analyses are often used to improve efficiency or effectiveness of optimizations, either by finding an approximate answer when the optimal answer would be prohibitively difficult to compute or by focusing the optimization on realistic behavior to increase optimization potential. The next subsection gives an introduction to profiling, an approach that also aims at increasing optimization potential.

2.2.2 Profiling

In many cases, programs do not behave as bad as the compiler expects. Hence, gathering information about the run-time behavior, called *profiling*, can be employed to focus optimization efforts on expectable behavior. To be used by compiler optimizations, gathered run-time information must be fed back to a static compiler, which is called *feedback directed optimization* (*FDO*). That is, the program has to be compiled first without run-time information, at the same time instrumenting the IR of the program to collect and emit relevant information. Then the binary must be executed sufficiently many times with typical input data for each execution to capture representative run-time behavior. This is usually the most time-consuming step because each execution of the instrumented program additionally involves to analyze and to store necessary run-time values and may introduce complex calculations with a substantial execution time overhead. Afterwards, gathered information is available to direct optimizations. As a result, profile-guided compilation is able to achieve noticeable performance improvements, though having the penalty to execute instrumented code and to re-compile the program after its execution. To mitigate the overhead, profile collection can be limited to an one-off execution run, though having the drawback to gather only a single run-time behavior, not expectable realistic behavior. In general, it works well only when the actual run-time program tendencies match those collected during profiling [Smi00]. Thus, profiling is strongly dependent on the input data set(s) of the execution run(s).

Integrating a beforehand machine learned model into the compiler that holds observations from several profiling runs and relates them to static code features, as we do, eliminates the need for profiling at compile time and has the advantage of always focusing optimizations on more realistic behavior, not on one single behavior as it is the case for a single profiling run. In the next section, we introduce the *Machine Learning* techniques that are relevant to our thesis.

2.3 Machine Learning

Machine Learning (*ML*) has been central to artificial intelligence research from the beginning [Tur50] and can be used to automatically infer information from a series of observations. The key advantage of ML techniques is their ability to find relevant information in a high-dimensional space. In the following we discuss the concepts of classification learning in Subsection 2.3.1 and regression modeling in Subsection 2.3.2, which are central for our approach. Then, we characterize quality measures of the precision of machine learned predictors in Subsection 2.3.2.

2.3.1 Supervised Classification Learning

In this subsection, we introduce classification learning, for which several algorithms have been proposed. For classification learning, each observation is described by its properties, or features, and is categorized concerning a set of given classes. The aim is to build a model that best explains the relationship from features to classes for the training data. More formally (see [Alp10] as reference), we have d-dimensional feature vectors $\mathbf{x}$ as input examples (observations). Given k classes denoted as C_i with $i = 1, \ldots, k$, an input example belongs to exactly one of them. Hence, the *training set* containing N examples is of the form

$$X = \{(\mathbf{r}^t, \mathbf{x}^t) | t = 1, \ldots, N\}$$

where t indexes different examples and $\mathbf{r}$ has k dimensions with

$$r_i^t = \begin{cases} 1 \text{ if } \mathbf{x}^t \in C_i \\ 0 \text{ if } \mathbf{x}^t \in C_j, j \neq i \end{cases} \quad (1 \leq i \leq k) \ .$$

In general, the learning algorithm should find a particular *hypothesis* $h \in H$ to approximate C as closely as possible. That is, h should minimize the *empirical error* which is the proportion of training instances where predictions of h do not match the required values given in X. This error of h is given by

$$Err(h|X) = \frac{1}{N} \sum_{t=1}^{N} \text{loss}_1(h(\mathbf{x}^t) \neq \mathbf{r}^t)$$

where $\text{loss}_1(a \neq b)$ is 1 if $a \neq b$ and is 0 if $a = b$. If $Err(h|X) = 0$, h is equal to C.

For classification, the aim is to learn the boundary separating the instances of one class from the instances of all other classes. In the following, we introduce two basic learning techniques, first *classification tree learning* and afterwards the *Naïve Bayes* approach.

Classification Tree Learning

Here, we consider classification tree learning [BFOS84], which is one of the most popular methods in data mining [Ste09]. It has become popular because of its clear interpretation and its ability to provide a good fit in many cases [GLM04]. The advantage compared to other methods like *neural networks* (see [Wan03]) or *nearest neighbors* (see survey paper [Ind04]) is that the constructed predictors have concise representations (decision trees). Thus, once trained, making predictions takes a negligible amount of time. Moreover, knowledge extraction from decision trees for explanation about the classification process is provided. That is, the decision tree can be inspected to determine which features are important for classification.

For classification tree learning, a k-class classification problem can be seen as k two-class problems. The training examples belonging to C_i are the positive instances of hypothesis h_i and the examples of all other classes are the negative instances of h_i. Hence in a k-class problem, we have k hypotheses to learn such that

$$h_i(\mathbf{x}^t) = \begin{cases} 1 \text{ if } \mathbf{x}^t \in C_i \\ 0 \text{ if } \mathbf{x}^t \in C_j, j \neq i \end{cases} \quad (1 \leq i \leq k) \ .$$

Classification trees are composed of *leaf nodes* that define the classes and of internal *decision nodes* that split the input space in two. Each decision node m implements a test function $f_m(\mathbf{x})$ with discrete outcomes labeling the branches. The test is a comparison $f_m(\mathbf{x})\colon x_j \geq w_m$ where x_j is a dimension of the feature vector $\mathbf{x}$ and w_m is a suitable chosen threshold value. Thus, learning the hypotheses reduces to the problem of determining these threshold values. The quality of a split is quantified by an *impurity measure*. A split is pure if after the split, for all branches, all the instances choosing a branch belong to the same class C_i, and all the instances choosing the other branch do not belong to C_i. Given an arbitrary but fixed node m, N_m is the number of training instances reaching node m (for the root node, it is N) and N_m^i of N_m belong to class C_i. On condition that an instance $\mathbf{x}$ reaches node m, the estimate for the probability of class C_i is

$$P(C_i|\mathbf{x}, m) \equiv p_m^i = \frac{N_m^i}{N_m} \ .$$

Node m is pure if p_m^i for all i are either 0 or 1. It is 0 when none of the instances reaching node m are of class C_i, and it is 1 if all such instances are

of C_i. If the split is pure, there is no need for further splitting, but a leaf node can be added that defines the class for which p_m^i is 1. Otherwise, one possible function to measure impurity is *entropy* defined as

$$\iota_m = -\sum_{i=1}^{k} p_m^i \log_2 p_m^i$$

where $0 \log 0 \equiv 0$ [Qui86]. Entropy in information theory specifies the minimum number of bits needed to encode the classification accuracy of an instance. It can be considered to be a measure of the uncertainty of an unknown result whose possible outcomes have certain probabilities. Intuitively, suppose the probability of an outcome of a binary decision is $p = 0$. At this probability, the outcome is certain never to occur, and so there is no uncertainty, leading to an entropy of $-0 \log_2 0 - 1 \log 1 = 0$. Vice versa, if the probability is $p = 1$, there is again no uncertainty and the entropy is $-1 \log_2 1 - 0 \log_2 0 = 0$. If the probability is $p = 0.5$ for the outcome of a binary decision, the uncertainty is at a maximum. In this case, the entropy ι takes its maximal value of 1. Intermediate values fall between these extreme cases. When there are $k > 2$ classes, the same discussion holds and the largest entropy is $\log_2 k$ when $p^i = 1/k$ $(i = 1, \ldots, k)$.

If the node m is not pure, then the instances should be split to decrease impurity, and there are multiple possible dimensions in the feature vector on which the split can happen. Among all, we look for the split that minimizes impurity after the split because we want to generate the smallest tree. Assumed at node m, N_{mj} of N_m instances take branch j (these are $\mathbf{x}^t$ for which the test $f_m(\mathbf{x}^t)$ returns outcome j). Then given that at node m, the test returns outcome j, the estimate for the probability of class C_i is

$$P(C_i|\mathbf{x}, m, j) \equiv p_{mj}^i = \frac{N_{mj}^i}{N_{mj}}$$

and the total impurity after the split is given as

$$\iota'_m = -\sum_{j=1}^{2} \frac{N_{mj}}{N_m} \sum_{i=1}^{k} p_{mj}^i \log_2 p_{mj}^i \quad .$$

To be able to calculate p_{mj}^i one needs to know the threshold value w_m for that node to implement the test function $f_m(\mathbf{x})$. There are $N_m - 1$ threshold values needed to separate N_m data points. There is no need to test for all (possibly infinite) threshold values between two data points. It is enough to test, for example, at halfway between points and take the best in terms of purity. For all dimensions d in the feature vector, and for the threshold variables w_m of all decision nodes m, the learning algorithm calculates the impurity and chooses the one that has the minimum entropy. Then tree construction continues recursively for all the branches that are not pure, until purity is maximal. When there is noise, growing the tree until it is purest may result in a large tree. To alleviate this, tree construction ends when nodes become pure enough, namely, when the impurity ι_m at node m is below a certain threshold τ_ι. Then, the set of training instances that reach node m is not split further, but a leaf

node is created and is labeled with the class having the highest estimate for the probability p^i_{mj}. This implies that we do not require p^i_{mj} to be exactly 0 or 1 but close enough, with a threshold τ_ρ.

Once trained, the model can be used to classify new observations and to automatically generate an executable heuristics. Breiman [Bre01] has proposed a more sophisticated learning technique based on classification trees called *Random Forest*. The *Random Forest* technique grows many classification trees where at each tree only a random subset of features is used. To classify a new observation, the forest yields the majority of votes over all trees as classification. We apply the *Random Forest* to derive predictions of expectable run-time behavior of applications, namely loop iteration counts and recursion frequencies of functions, based on feature vectors that represent static properties of the source code of the application. The idea of the heuristics is to focus on typical program behavior instead of considering all possibilities, as it is done by most static analyses used in compilers. This provides the compiler of our framework with intelligence about the dynamic program behavior. We use this information for a precise cost estimation to improve the mapping of concurrent applications to parallel architectures.

Naïve Bayes

The Naïve Bayes technique uses the *Bayes' theorem* to learn a hypothesis h, based on the assumption that the features $\mathbf{x}$ used for deriving a prediction are statistically independent of each other given the predicted value. The technique introduces the so-called *a posteriori* probability $P(h|X)$ of the hypothesis h given the training sample X to minimize the expected prediction error. Applying Bayes' theorem, this probability is given by

$$P(h|X) = \frac{P(h)P(X|h)}{P(X)} \tag{2.1}$$

where $P(h)$ is the *a priori* probability of the hypothesis h. $P(h)$ is the knowledge regarding the possible instantiations of h *before* looking at the training sample X. $P(X|h)$ is the *sample likelihood*, also called the *likelihood density function*, that denotes how likely the sample X is for the instantiation of h. $P(X)$ in the denominator of Equation (2.1), called the *evidence*, is the a priori probability of the training set that an observation $\mathbf{x}$ is seen. Because it does not depend on h and the values of the features in X are given, the denominator is effectively a constant that does not have to be computed.

The a posteriori probability $P(h|X)$ denotes the likelihood of h *after* looking at the sample. Hence the Naïve Bayes approach has to solve the following equation to find the most probable hypothesis given the training set X, i. e., to find the *maximum a posteriori* hypothesis h_{MAP}:

$$h_{MAP} = \arg\max_{h \in H} \frac{P(h)P(X|h)}{P(X)} = \arg\max_{h \in H} P(h)P(X|h) \ , \tag{2.2}$$

where arg max returns the argument that maximizes $P(h)P(X|h)$. Assuming equal probabilities of hypotheses h, Equation (2.2) reduces to the so-called *maximum likelihood* hypothesis h_{ML}, given by

$$h_{ML} = \arg \max_{h \in H} P(X|h) \ . \tag{2.3}$$

We use the Naïve Bayes technique as possible classifier for inner nodes of our decision tree based learning technique, which is an adaption of the *Predicting Query Run-time 2* (*PQR2*) algorithm, originally developed by Matsunaga and Fortes [MF10], to learn execution times of tasks (see Section 4.3.3).

2.3.2 Regression Modeling

Both regression and classification are supervised learning problems where solving the problem refers to learning the mapping from an input to the output. In case of regression modeling, the output is a numerical value and not a class code as it is the case for classification learning. More formally, (see [Alp10] as reference), we have d-dimensional feature vectors $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ as input examples (observations) and the associated output r of each observation, which is a number. Consequently, the training set containing N observations is of the form

$$X = \{(r^t, \mathbf{x}^t) | t = 1, \ldots, N\}$$

where t indexes different examples and $r^t \in \mathbb{R}$. Given the *Central Limit Theorem*[3] the output r^t is assumed to be normally distributed [Ric07]. If there was no noise, deriving the mapping from input to output would be to find the function $f(\bullet)$ that passes through the points r^t such that we have

$$r^t = f(\mathbf{x}^t)$$

which is called interpolation. In regression, there is noise added to the output of the unknown function

$$r^t = f(\mathbf{x}^t) + \eta$$

where $f(\bullet)$ is the unknown function, $f(\mathbf{x}) \in \mathbb{R}$ and η is random noise. The explanation for noise is that there are hidden variables that we cannot observe: $r^t = f^*(\mathbf{x}^t, \mathbf{z}^t)$ where $\mathbf{z}^t$ denote those hidden variables. The noise is assumed to be normally distributed with constant variance σ^2 and zero mean μ, i. e., $\mu = \mathrm{E}[\eta] = 0$. The aim of regression modeling is to find a model $g(\mathbf{x}|\Theta)$ that best approximates f^*, where $g(\bullet)$ is the model, $\mathbf{x}$ is the input, and Θ are the

[3] The *Central Limit Theorem* states that the sample mean of (a sufficiently large number of) independent and identically distributed random variables will be approximately normally distributed.

parameters that must be learned. The difference between the output of the model and the actual observed value r^t is called the *residual*[4].

In the following, $\hat{r}$ denotes the output of the model $g(\mathbf{x}|\hat{\Theta})$ given the current value of the parameters, $\hat{\Theta}$. Because r and $\hat{r}$ are numeric quantities, there is an ordering defined on their values and we can define a *distance* between values as error measure, which gives us more information than equal/not equal as used in classification. The function that computes this distance is called the *loss function* $L(r,\hat{r})$. The *approximation error*, or *loss*, is the sum of losses over the individual instances

$$Err(\hat{\Theta}|X) = \sum_{t=1}^{N} L(r^t, \hat{r}^t) \ .$$

Again, the learning approach is in principle the same as for classification. We assume a hypothesis class H for $g(\bullet)$ and the learning algorithm has to find a particular Θ^* that minimizes the approximation error:

$$\Theta^* = \arg\min_{\Theta} \ Err(\Theta|X)$$

where arg min returns the argument that minimizes $Err(\bullet)$.

In case of *linear regression*, given the feature vector $\mathbf{x} = (x_1, x_2, \ldots, x_d)$, we assume a linear model $g(\bullet)$ such that we have

$$g(\mathbf{x}|\Theta) = \boldsymbol{\theta} \cdot \mathbf{x} + \beta = \sum_{i=1}^{d} \theta_i x_i + \beta \tag{2.4}$$

where $\cdot$ is the dot product of two vectors. The parameters Θ that must be learned are the weighting vector $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_d)$ and the intercept β. The model $g(\bullet)$ defines the hypothesis class H and a particular value of Θ instantiates one hypothesis $h \in H$. Hence, learning the hypothesis h^* that approximates f^* as closely as possible is the problem of determining the parameters Θ^* that instantiate h^*. There exist different learning algorithms that differ in the techniques and loss measures they employ. In the following, we describe basic approaches to determine Θ^* for linear regression.

Ordinary Least Squares Estimation

One optimization technique is *Ordinary Least Squares Estimation* (*OLS*), where the loss function is the squared distance between the observed outputs and the outputs predicted by the model: $L(r,\hat{r}) = [r - \hat{r}]^2$. The learning

[4]The statistical error is the amount by which an observation differs from its expected value. The latter is the mean of the entire population, i. e., of all instances of $(\mathbf{x}, r)$ that are potentially observable. Because the whole population is not known in general, the expected value is typically unobservable, and hence the statistical error cannot be observed either. The residual, being the difference between the observed and predicted output, is an observable estimate of the unobservable statistical error.

goal for OLS is to minimize the sum of losses over the individual observations, which is called the *residual sum of squares* (*RSS*). Hence, the parameters Θ^* that have to be found are given by

$$\Theta^* = \arg\min_{\Theta} \text{RSS} = \arg\min_{\Theta} \sum_{t=1}^{N} [r^t - \hat{r}^t]^2 \ . \tag{2.5}$$

The value of Θ^* that minimizes this sum is called the OLS estimator for Θ. The Gauss-Markov theorem [ZM69] states that this OLS estimator is the *best linear unbiased estimator* of Θ, where best means giving the lowest possible empirical error of the estimate. Unbiased means that the expected value of the estimator is equal to the correct value of the parameters being estimated. However, OLS estimates are non-robust to outliers. Outliers are observations which do not follow the pattern of the other observations. Furthermore, OLS can give misleading results if the underlying assumptions of linear regression are not true; thus ordinary least squares is said to be not robust to violations of its assumptions.

Robust Linear Regression

For robust linear regression, one method of estimating parameters in a regression model that is less sensitive to outliers and to violations of underlying assumptions than the OLS estimator, is to use the *Least Absolute Deviations* (*LAD*) technique [NW82]. The method minimizes the sum of absolute errors, i. e., it aims at determining

$$\Theta^* = \arg\min_{\Theta} \sum_{t=1}^{N} |r^t - \hat{r}^t| \ .$$

Another technique of estimating parameters that is more computational complex is the *Iterated Reweighted Least Squares* (*IWLS*) method. IWLS uses a *weighted* least squares objective function to find the estimator for Θ:

$$\Theta^* = \arg\min_{\Theta} \sum_{t=1}^{N} w^t(\hat{r}^t)[r^t - \hat{r}^t]^2 \ . \tag{2.6}$$

This is done by *iteratively* solving a weighted least squares problem of the form

$$\Theta_{n+1} = \arg\min_{\Theta} \sum_{t=1}^{N} w^t(\hat{r}_n^t)[r^t - \hat{r}_n^t]^2$$

in each step n until the values for Θ converge. The prediction $\hat{r}_n^t$ is the output of the model $g(\bullet)$ given the current parameters $\hat{\Theta}_n$, i. e., $\hat{r}_n^t = g(\mathbf{x}^t|\hat{\Theta}_n)$. The working weights w^t can be obtained from the reciprocal of the error variance of $\hat{r}_n^t$ (see [Dob02] for details). In this way, the predictions which are less reliable (that is, the predictions with the larger variances) will have less influence

on the estimates. The initial value $\hat{\Theta}_0$ for the 0^{th} iteration step can be given by the OLS estimator.

k-Nearest Neighbor

The *k-Nearest Neighbor* (*k-NN*) estimator is a nonparametric estimation technique. That is, it does not aim at determining the parameters Θ to learn the relationship from features to the associated output. Instead, k-NN is based on closest training examples in the feature space, the *neighbors*, where closest is defined by a distance measure between observations in the training set and the feature vector for which the prediction of the output is required. Given the distance measure d and a vector $\mathbf{x}$, we define an ordering relation $d_1(\mathbf{x}) \sqsubseteq d_2(\mathbf{x}) \sqsubseteq \cdots \sqsubseteq d_N(\mathbf{x})$ where $d_1(\mathbf{x})$ is the distance to the nearest observation in the training set, $d_2(\mathbf{x})$ is the distance to the next nearest, and so on. A usual distance measure between feature vectors $\mathbf{x}$ with continuous variables as features, i. e., $\mathbf{x} \in \mathbb{R}^d$, is the *Euclidean distance* given by

$$d_{Euc}(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{d} [x_i - y_i]^2} \ .$$

Then we define $d_1(\mathbf{x}) = \min_t d_{Euc}(\mathbf{x}, \mathbf{x}^t)$, and if i is the index of the nearest observation, namely $i = \arg\min_t d_{Euc}(\mathbf{x}, \mathbf{x}^t)$, then $d_2(\mathbf{x}) = \min_{t \neq i} d_{Euc}(\mathbf{x}, \mathbf{x}^t)$, and so forth. Using the Euclidean distance, the ordering relation $\sqsubseteq$ equals $\leq$ from the domain $\mathbb{R}^2 \to \mathbb{B}$.

The model parameter k is a user-defined constant integer, which is much smaller than the sample size N, and defines the number of neighbors taken into account. For regression modeling, the predicted output is the average of the associated outputs of its k nearest neighbors. If observations in the training set contain noisy or irrelevant features, or if the feature scales are not consistent with their importance, the accuracy of the k-NN algorithm can be degraded. Hence, it can be useful to scale features or to weight each associated output of the neighbors, so that the nearer neighbors contribute more to the average than the more distant ones. Common scaling functions for a feature x_i are the n^{th} power of x_i (n is a small integer) or the logarithm of x_i to base e, and common weighting schemes are *inverse distance weighting* (with $\frac{1}{d}$) or *exponential distance weighting* (with e^{-d}) [NBvB+06].

The k-NN technique is independent of the underlying distribution on the observations and associated outputs. Thus, it is not sensitive to violations of assumptions about the distribution. Cover and Hart [CH67] have shown that the k-NN algorithm has some strong consistency results. As the amount of training examples approaches infinity, the algorithm is guaranteed to yield an error rate no worse than twice the *Bayes error rate*, which is the lowest achievable error rate given the distribution of the data. However, since the training phase of the algorithm consists only of storing the feature vectors and associated outputs, the technique has the drawback of being computationally expensive during prediction, especially when the size of the training set grows.

Furthermore, the technique requires in that case a large memory consumption to store the training set.

Support Vector Machine

The *Support Vector Machine* (*SVM*) method was initially proposed as a universal approach for solving classification problems and was later extended to regression problems (see [Gun98] as reference). The intuition of SVMs for classification is to separate the space of possible outputs using hyperplanes based on a small subset of the training data instead of predicting the output from the entire training set as it is the case for the k-NN technique. The set of vectors is said to be optimally separated by the hyperplane if it is separated without error and the distance between the closest vector to the hyperplane, called the *margin*, is maximal. The hyperplane can be constructed by solving a constrained quadratic optimization problem in terms of a subset of the feature vectors in the training set that lie on the margin. These training patterns, called the *support vectors*, carry all relevant information about the problem.

For linear regression, Vapnik [Vap95] proposed the so-called ε-*insensitive* loss function given by

$$L_\varepsilon(r,\hat{r}) = \begin{cases} 0 & \text{if } |r-\hat{r}| < \varepsilon \\ |r-\hat{r}| - \varepsilon & \text{otherwise} \end{cases} .$$

This loss function ignores errors that are smaller than a certain threshold $\varepsilon > 0$, thus creating a tube around the true output so that the loss is zero if the predicted value is within the tube, while the loss is the magnitude of the difference between the predicted value and the radius ε of the tube if the predicted value is outside the tube. When $\varepsilon = 0$, we get the loss function used for the LAD technique.

Based on this loss function, Vapnik [Vap95] introduces the regression model

$$\begin{aligned} g(\mathbf{x}|\Theta) &= \sum_{t=1}^{N} (\tilde{\alpha}_t - \alpha_t)(\mathbf{x}^t \cdot \mathbf{x} + 1) + b \\ &= \sum_{t=1}^{N} (\tilde{\alpha}_t - \alpha_t)\mathbf{x}^t \cdot \mathbf{x} + \sum_{t=1}^{N} (\tilde{\alpha}_t - \alpha_t) + b \end{aligned} \tag{2.7}$$

that explicitly uses the training examples within the dot product $\cdot$ and introduces the *Lagrange multipliers* $\tilde{\alpha}$ and α as multiplicative constants with $\tilde{\alpha}, \alpha > 0$. Each $\tilde{\alpha}_i$ will be nonzero if the corresponding observed value is below the tube and α_i will be nonzero if the observed value is above the tube. Since an observed value can not be simultaneously on both sides of the tube, either $\tilde{\alpha}_i$ or α_i will be nonzero, unless the value is within the tube, in which case both constants will be zero. The feature vectors r^i corresponding to a nonzero $\tilde{\alpha}_i$ or α_i are termed the support vectors. Then, the $2N+1$ values of Θ for $\tilde{\alpha}_i$,α_i, and b must be determined from the N training vectors. The author has shown that this problem can be solved by maximizing the objective

function

$$F(\tilde{\alpha},\alpha)=\sum_{t=1}^{N}[\tilde{\alpha}_t(r^t-\varepsilon)-\alpha_t(r^t+\varepsilon)]-\frac{1}{2}\sum_{s,t=1}^{N}(\tilde{\alpha}_s-\alpha_s)(\tilde{\alpha}_t-\alpha_t)(\mathbf{x}^s\cdot\mathbf{x}^t+1)$$

subject to constraints

$$\begin{aligned}&0<\tilde{\alpha}_i<C \text{ and } 0<\alpha_i<C\ (i=1,\ldots,N)\\&\sum_{t=1}^{N}\tilde{\alpha}_t=\sum_{t=1}^{N}\alpha_t\end{aligned}\tag{2.8}$$

where C is a user-defined constant. Hence, solving $\tilde{\alpha}^*,\alpha^*=\arg\max_{\tilde{\alpha},\alpha}F(\tilde{\alpha},\alpha)$ subject to the constraints above determines the values $\tilde{\alpha}^*,\alpha^*$ that minimize the approximation error. That is, considering Equation (2.4) as our general model for linear regression and comparing it with Equation (2.7), the weighting vector $\boldsymbol{\theta}^*=(\theta_1^*,\theta_2^*,\ldots,\theta_d^*)$ of parameters Θ^* is given by the vector

$$\mathbf{w}^*=\sum_{t=1}^{N}(\tilde{\alpha}_t^*-\alpha_t^*)\mathbf{x}^t\tag{2.9}$$

and β^* equals b^* because given the constraint in Equation (2.8), $\sum_{t=1}^{N}(\tilde{\alpha}_t^*-\alpha_t^*)$ is zero. The value of b^* can be computed as follows (see [BPP07])

$$\begin{aligned}b^*&=r^t-\mathbf{w}^*\cdot\mathbf{x}^t-\varepsilon \quad \text{for } \tilde{\alpha}_t^*\in(0,C)\\\text{or } b^*&=r^t-\mathbf{w}^*\cdot\mathbf{x}^t+\varepsilon \quad \text{for } \alpha_t^*\in(0,C)\ .\end{aligned}$$

Instead of using an arbitrary support vector $\mathbf{x}^t$ to compute b^*, the average over all support vectors can be used for numerical stability.

Decision Tree

Decision trees, as introduced for supervised classification learning in Subsection 2.3.1, can also be used for the regression problem. In contrast to classification where the leaf nodes of the tree define the classes, for regression the leaf nodes represent decisions on the numerical target. Using the approach of Breiman et al. [BFOS84], a so-called *regression tree* is constructed in almost the same manner as a classification tree, except that the impurity measure that is appropriate for classification is replaced by a quality measure appropriate for regression. Given an arbitrary but fixed node m, let X_m be the subset of the training set X containing N_m training instances that reach node m (see [Alp10] as reference). We define a function that indicates whether a feature vector $\mathbf{x}$ reaches node m as

$$b_m(\mathbf{x})=\begin{cases}1 \text{ if } \mathbf{x}\in X_m\\0 \text{ otherwise}\end{cases}\ .$$

Using this definition, N_m is given by $N_m=\sum_{t=1}^{N}b_m(\mathbf{x}^t)$. In regression, the quality of a split is measured by the mean square error from the estimated

value $\overline{r}_m$ at node m, where the estimated value is the sample mean of the required outputs of training instances reaching the node. Hence,

$$\overline{r}_m = \frac{1}{N_m} \sum_{t=1}^{N} b_m(\mathbf{x}^t) r^t$$

and from this, we obtain the mean square error E_m via

$$E_m = \frac{1}{N_m} \sum_{t=1}^{N} [r^t - \overline{r}_m]^2 b_m(\mathbf{x}^t) \ . \tag{2.10}$$

Equation (2.10) corresponds to the variance at node m. If at a node, the error is acceptable, namely below a certain threshold τ_m, then a leaf node is created and it stores the value $\overline{r}_m$. If the error is not acceptable, the set reaching node m is split further such that the sum of the errors in the branches is minimal. As in classification, we look at each node for the attribute that minimizes the error and then we continue recursively. More formally, let X_{mj} be the subset of X_m taking branch j and b_{mj} indicates whether vector $\mathbf{x} \in X_m$ takes branch j:

$$b_{mj}(\mathbf{x}) = \begin{cases} 1 \text{ if } \mathbf{x} \in X_{mj} \\ 0 \text{ otherwise} \end{cases} \ .$$

Then, the estimated value at branch j is given by

$$\overline{r}_{mj} = \frac{1}{N_{mj}} \sum_{t=1}^{N} b_{mj}(\mathbf{x}^t) r^t$$

where $N_{mj} = \sum_{t=1}^{N} b_{mj}(\mathbf{x}^t)$, and the error E'_m after the split is

$$E'_m = \frac{1}{N_m} \sum_{j} \sum_{t=1}^{N} [r^t - \overline{r}_{mj}]^2 b_{mj}(\mathbf{x}^t) \ . \tag{2.11}$$

The drop in error for any split is given as the difference between Equation (2.10) and Equation (2.11). We look for the split such that this drop is maximal, or equivalently, where Equation (2.11) takes its minimum. Instead of taking the mean at a leaf as constant value, we can also do a linear regression fit over the instances choosing the leaf:

$$g_m(\mathbf{x}|\Theta_m) = \boldsymbol{\theta}_m \cdot \mathbf{x} + \beta_m \ .$$

This makes the estimate in a leaf dependent on $\mathbf{x}$ and generates smaller trees because with the use of $g_m(\bullet)$ instead of $\overline{r}_m$, the error should be sooner below the threshold τ_m. However, there is the expense of extra computation at a leaf node.

A recently developed approach based on a decision tree is the technique of Gupta et al. [GMD08]. It proposes the Predicting Query Run-time (PQR) tree that can be used for classification to predict at run-time the range of

the execution time of database queries. Matsunaga and Fortes [MF10] have extended the PQR tree approach to the regression problem by allowing the leaf nodes of the tree to select the best regression method. We describe this related work in more detail in Section 3.4.

The above mentioned basic approaches for linear regression differ in the learning techniques and the loss measures they employ. We use regression modeling to learn execution times of tasks with feature vectors that hold static properties of the source code of the concurrent application to be mapped. Based on this, a heuristics is automatically generated as a static predictor for the time that a task will consume at run-time of the application. This information is used by our framework for a precise cost estimation to improve the allocation of tasks to *Processing Elements* (*PEs*) in order to reduce the total execution time of the concurrent application. To determine the best suitable basic approach for linear regression, we have applied each approach to learn execution times and we have evaluated the precision of the resulting predictors. As precision measure to asses and compare their quality, we use the metrics that we describe in the next subsection.

2.3.3 Predictor Precision

To evaluate the precision of a machine learned predictor, it is applied to data for which the correct outcomes are known. Then, correct and predicted outcomes can be compared. For realistic results, predictors should be applied to unseen data for evaluation. As precision measure, the average deviation between predictions and correct outcomes can be used (mean absolute error). Additionally, considering the correlation between predictions and correct outcomes helps to decide whether a relationship was actually learned. For a closer view on classification learning, the Δk-Accuracy, which denotes the fraction of predictions with a maximal absolute deviation of k classes, provides further details.

In the previous sections, we have introduced how applications are compiled, how properties of the program can be analyzed, and how static information about these properties can be related to the dynamic run-time behavior of the program via *Machine Learning*. In the following section, we describe parallel programming models used to implement concurrent applications. Next, we describe in Section 2.5 parallel architectures on which these applications can be executed and in Section 2.6 the challenge of mapping concurrent applications to parallel architectures.

2.4 Parallel Programming Models

A parallel programming model is an abstract *parallel processing model* used to express parallel programs, which can be compiled and executed. It abstracts from hardware and describes a parallel computing system in terms of the se-

mantics of the programming language or programming environment (see [RR10] as reference). The types of models for parallel processing according to [HR92] differ in their level of abstraction from the target architecture. The four basic types are *machine models*, *architectural models*, *Models of Concurrent Computation* (*MoCCs*), and *programming models*. Machine models are at the lowest level and consist of a description of hardware and operating system. Architectural models include properties of the parallel execution mode of single instructions, i. e., the modes *Single Instruction Multiple Data* (*SIMD*) or *Multiple Instruction Multiple Data* (*MIMD*) corresponding to Flynn's taxonomy [Fly72]. Additionally, they include information about the interconnection network and the memory organization of the parallel target platform (e. g., shared versus distributed memory), and about types of processing (e. g., synchronous versus asynchronous). The MoCC offers a more formal model that provides cost functions reflecting the time needed for the execution of a program on the resources of the target platform described by the architectural model[5]. Thus, an MoCC provides an analytical method for designing and evaluating algorithms.

Parallel programming models are at the highest level of abstraction and thus are not architecture specific. They specify the programmer's view on parallel computing systems by defining how the programmer can implement an algorithm. There are several criteria by which the models can differ such as the following:

- **Problem Decomposition** Problem decomposition refers to the way in which the parallelly executable parts of concurrent applications are formulated. These parts can be specified at different levels of granularity [OB07]. The lowest level of granularity is *instruction level parallelism* where the instruction stream of machine code is parallelized, such as load, store, and *Arithmetic Logic Unit* (*ALU*) operations. Machine code instructions are not visible in the source code of high level programming languages and so cannot be manipulated by the programmer.

  The *data level*, also referred to as the concept of *parallel loops*, focuses on performing operations on a data set. Sequences of operations are collectively performed on the same data structure, however, each sequence works on a different partition of the same data structure. Each sequence is on a small scale a BB of the IR or on a larger scale a loop iteration and the data structure then is partitioned according to the current loop index. This level is exploited in sequential programs by optimizing compilers via loop parallelization techniques or by the programmer via compiler directives that allow to specify the distribution and alignment of data.

  The *task level*, also known as the function level is commonly regarded as the first potential of parallelism brought into attention in many applications which can result in considerable performance gains [OB07]. A task

[5] For example, the *Random Access Machine* (*RAM*) model is a MoCC that describes sequential computing on the von Neumann architecture and the *Parallel Random Access Machine* (*PRAM*) model is the generalization of the RAM model used for parallel processing.

is a subset of the control flow of a program that is executed by a processor. Ostadzadeh and Bertels [OB07] additionally define higher levels of granularity, the *process* or *thread level* parallelism, but usually they are subsumed under the task level. At process level, each process has its own separate address space. To communicate, a process must send and explicitly receive a message. At the thread level, the threads share an address space and therefore can additionally communicate through shared variables. Using this categorization, a process can consist of several threads which share a common address space whereas each process works on a different address space.

- **Organization of Address Space** An important criterion is the organization of the address space because it has a significant influence on the information exchange between the parallelly executable tasks of a program. The address space can be *shared* or *distributed*, but also a hybrid organization combining both features exists. For the shared address space, there is one global memory that can be accessed from all parts of the program. In that case, information exchange can be performed by direct read or write accesses to shared variables. In case of a distributed address space, each parallelly executable task of a program has its own local memory. Therefore, information exchange must be performed by additional message-passing operations.

- **Execution Mode of Parallel Parts** The execution mode of the parallel parts of a program defines how they are executed. One categorization is whether the parts are executed *synchronously* or *asynchronously*. Further, the execution mode determines whether the parts are identical. For example, the execution of parallel tasks is categorized into *Single Program Multiple Data* (*SPMD*) or *Multiple Program Multiple Data* (*MPMD*), which are subcategories of the MIMD execution mode. In SPMD mode, multiple instances of the same program are simultaneously executed with different input data and at independent points (rather than in the lockstep that SIMD imposes). SPMD was proposed in 1984 by Darema [Dar01] for highly parallel machines. The first SPMD standard was *Parallel Virtual Machine* (*PVM*). The current de facto industry standard is *Message Passing Interface* (*MPI*). In MPMD mode, at least two different programs are independently executed.

- **Exchange of Information** Information exchange relates to the mechanisms by which parallel tasks are able to communicate with each other. The most common forms of interaction are *shared variables* and explicit communication via *message passing*. Using shared variables which can be asynchronously accessed by parallel tasks for reading and writing requires protection mechanisms such as locks, semaphores and monitors to synchronize and control concurrent access. In case of message passing, the communications can be asynchronous or synchronous. Forms of messages include remote method invocation, signals, and data packets. By waiting for messages, the parallel tasks can also synchronize.

- **Specification of Parallelism** The parallelism of an application is either explicitly or implicitly defined. If the parallelism is implicitly defined, the program has to be parallelized, e. g., by the compiler. In this thesis, we assume explicitly defined parallelism because we focus on improving the mapping of concurrent applications to parallel architectures, not on the extraction of parallelism in an application, which is another research area.

There is a large number of different possibilities for combination of these criteria. The main task-level parallel programming models [Fos95, PG08] in common use are:

Shared Memory Model In the shared memory programming model, a single common memory is shared by all parallel tasks of an application. Because threads are defined via their property of sharing an address space, this model is also called the *thread* model or the *multithreading* model. Common low-level *Application Programming Interfaces* (*APIs*) for threads are Boost Threads [WE12] and the POSIX threads API, usually abbreviated as Pthreads library and specified by the IEEE POSIX 1003.1 standard [IEE04]. These APIs use mutual exclusion locks and conditional variables for establishing communication and synchronization between threads. *Threading Building Blocks* [Rei07] and *Cilk* [Cil01] provide parallel abstractions that take care of many of the low-level aspects, such as thread management and synchronizing shared data.

A shared memory API that abstracts from threads is the industry standard *Open Multi-Processing* (*OpenMP*) [OMP11]. OpenMP extends the languages with parallel directives that can be added to loops or code sections for instructing the compiler to parallelize these regions. OpenMP hides the details like work load partitioning, communication and synchronization. A drawback of OpenMP is that it does not provide the capabilities for managing the affinity of tasks to PEs.

Message Passing Model Message passing is used for large systems where the memory is distributed among a network of PEs and where each PE has direct access only to its local memory. In this model, data is moved from the address space of one process to that of another process through cooperative operations in both processes.

Message Passing Interface (*MPI*) [MPI12] is the technology of choice for constructing scalable parallel programs within this model, and its ubiquity means that no other technology can beat it for portability [GKT+09]. Each MPI process is an instance of the same MPI program, executing its own code in an MIMD style. That is, MPI programs are executed in SPMD mode. Typically, each MPI process executes in its own address space, although shared memory implementations of MPI are possible [MPI12]. Hence, MPI may also be used in shared memory architectures.

Distributed Shared Memory Model The distributed shared memory model has been developed to keep the advantages of shared memory programming in

distributed systems without a common shared memory. This model provides an abstraction of shared memory in a distributed memory environment by hiding the distributed nature of the memory from the user. Thereby, it offers the user the transparent access to local and remote memory by hiding the underlying communication between the different nodes.

Unified Parallel C (*UPC*) [UPC05] is a parallel extension of the C standard that realizes this model. UPC follows the SPMD execution mode. That is, the threads are instances of the same program, but each thread operates on different data. The threads are placed on the individual PEs at startup and do not migrate during run-time of the program. The main goal of this placement is to minimize the communication overhead between two cooperating threads, but the programmer has to choose a good allocation.

Hybrid Model When two programming models are used in one application, the application is said to be parallelized with a hybrid programming model. For example, a hybrid model can combine message passing and shared memory access, which may be realized using MPI together with threads or with OpenMP.

The two major parallel programming models are the message passing model for which MPI is the de facto standard and the shared memory model for which OpenMP is the de facto standard [KKJ+08]. The study of Jost et al. [JJaMH03] compares the performance of different implementations of the same application. The implementations employed either OpenMP as shared memory programming model, MPI as message passing model, or two hybrid programming models. Which programming model is the best depends on the nature of the given problem, on available software, and on the underlying architectural model. When using a high-speed interconnection network or shared memory, the pure MPI paradigm turned out to be most efficient, while for slower networks the benefit of hybrid implementations was visible. Rabenseifner et al. [RHJ09] also compare these programming models and observe that the network topology has a significant impact on the performance. They show that a hybrid model can be superior to the other because it can reduce communication costs. Hence, they demand that the application itself should be aware of the underlying topology and should apply the adequate actions depending on the underlying hardware. Mallón et al. [MTT+09] compare MPI, OpenMP and UPC with each other. They found that programming language features can ease parallel programming, but that they cannot hide the underlying communication costs.

With this thesis, we aim at improving the mapping of concurrent applications that are expressed in one of the parallel programming models described above. Any programming model can theoretically be implemented on any underlying hardware. For example on a distributed memory architecture a shared memory model can be established via *virtual shared memory*. Vice versa, on a shared memory architecture a distributed memory model can be used, such as MPI programs running on a multi-core processor. In the next section, we

introduce different architectural models onto which a concurrent application can be mapped.

2.5 Parallel Architectures

The previous section has introduced parallel programming models that can be implemented on underlying hardware. Since parallel programming models abstract from the actual hardware architecture, they define the user's view of the target architecture, i. e., the logical organization. In this section, we describe the physical organization, i. e., actual underlying hardware architectures. Parallel architectures can be classified according to the level at which the hardware supports parallelism and according to the connection of their parallel *Processing Elements* (*PEs*). The following list presents the main classes of parallel architectures available today.

Multi-Core Processor A multi-core processor is a processor with multiple *Central Processing Units* (*CPUs*), called cores. The CPUs are located on a single integrated circuit die or on multiple dies in a single chip carrier. Commonly, these CPUs are connected via a system bus, a ring, a two-dimensional mesh, or a crossbar switch. The CPUs share memories and may have local as well as shared caches.

Each CPU may have several functional units working in parallel: the *Arithmetic Logic Unit* (*ALU*), which performs arithmetic and logical operations between values in two's complement binary number representation, the *Floating-Point Unit* (*FPU*) for operations between numbers in floating-point representation, the *Load/Store Unit* (*LSU*) that loads data from memory or stores it back to memory from registers, and the *Control Unit* (*CU*), which extracts instructions from memory and decodes and executes them, calling other functional units when necessary. Although the functional units of a CPU work in parallel, we do not term them a PE because they only exploit instruction level parallelism and not task level parallelism. That is, the lowest granularity for PEs onto which a task can be mapped are the cores of a multi-core processor. Hardware examples are the *AMD Phenom*, the *AMD Opteron*, and Intel's *Core i3, i5, i7* families of multi-core processors as used in today's personal computers.

Symmetric Multiprocessor A *symmetric multiprocessor* (*SMP*) is a system with centralized shared memory and two or more tightly coupled *homogeneous* processors running independently under a single operating system. An SMP can be composed of thousands of identical processors. The processors commonly are connected using a system bus or a crossbar switch.

Today's used SMP servers are usually based on multi-core architectures like the *AMD Opteron* or Intel's *Xeon* and *Itanium 2* processors. Because of the physical and electrical constraints, larger SMP systems with more than 16 processors must use a hierarchy of interconnection mechanisms [Che05]. A large SMP tends to be a collection of smaller systems, each of which is itself

an SMP, i. e., a group of processors plus memory called *module*. Such a system allows any processor to access any memory, although there is a difference in the access time to within-module memory and to memory in another module. When a processor accesses its own memory, it will see a much shorter access time than when it accesses memory in a distant module. Such architectures are therefore referred to as *Non-Uniform Memory Access* (*NUMA*) systems. This non-uniformity of memory access times can have a major effect on system behavior and on the software running on it. Hardware examples are the *IBM X-Architecture*® servers composed of Opteron or Xeon processors and the *Cray T3E*™ supercomputer, a massively parallel processing system with up to 2048 processors.

Specialized Parallel Hardware In the past, the processor frequency closely followed Moore's law, which states that you can integrate twice as many components onto an integrated circuit every 18 months for the same cost. However, physical constraints have stopped and even slightly reversed this exponential increase of processor frequency. A key physical constraint is power density, often referred to as the *power wall* [BDH+10]. The power density in processors has already surpassed that of a hot plate, and is approaching the physical limit of what silicon can tolerate with current cooling techniques. Another factor why advances in technology and frequency scaling do not result in considerable performance improvements is the growing disparity of speed between PEs and the memory, called the *memory wall*. Now, with these additional constraints, the primary method of gaining extra performance out of computing systems is to introduce additional specialized hardware tailored to execute specific computations very fast.

One example of a specialized electronic circuit is the *Graphics Processing Unit* (*GPU*), used to accelerate the memory-intensive work of texture mapping, rendering polygons, and geometric calculations for computations related to manipulating computer graphics. More recent developments introduce *General Purpose GPUs* (*GPGPUs*) as a modified form of a streaming processor. Opposed to being hard wired solely to do graphical operations as it is the case for GPUs, the GPGPUs can be used for many types of parallel computations including ray tracing, computational fluid dynamics and weather modeling. However, they are only suited to high-throughput type computations that exhibit data parallelism to exploit its wide vector width SIMD architecture. That is, typically the performance advantage is only obtained by running the single active program simultaneously on many example problems in parallel [HB07]. A recent survey of Owens et al. [OLG+07] summarizes general-purpose application development on GPGPUs. Modern GPU cards include up to 32 streaming processors, for example the *GeForce GTX 590* developed by NVIDIA. That is, the graphic card has 32 PEs onto which tasks can be mapped.

Another example of a specialized hardware is the *Field Programmable Gate Array* (*FPGA*). FPGAs contain programmable logic components called *logic blocks* and a hierarchy of reconfigurable interconnects between the blocks. Depending upon granularity, the logic blocks implement functionalities ranging from small special functions to complex processor cores called *soft micropro-*

cessors. Modern FPGAs also include memory elements as logic blocks and integrate existing embedded microprocessors with related peripherals. Implementing more than one soft microprocessor or integrating a multi-core microprocessor within the FPGA results in a configurable multi-core processor. For instance, the *Xilinx Zynq-7000* family of FPGA boards and the *Altera Arria V* FPGA board include dual-core ARM processors.

Multi-Processor System-on-Chip (*MPSoC*) An MPSoC integrates multiple PEs and all other components of the electronic system like memory, peripherals, and external interfaces on the same chip. All functional blocks of an MPSoC are connected by a bus such as the *Advanced Microcontroller Bus Architecture* (*AMBA*) bus from ARM, a point-to-point infrastructure, or a *Network-on-Chip* (*NoC*). For efficient communication between the PEs, a NoC has several advantages over others, such as scalability and shorter wires, which minimizes power consumption, and it is the most promising candidate that interconnects the PEs through a configurable mesh of on-chip connections [HWC04].

As mentioned above, increasing the processor frequency is limited by physical constraints. Hence, the need for high performance and highly reactive systems that interact with other environments like multimedia systems requires the integration of specialized computational resources. Therefore, the PEs of MPSoCs become more and more heterogeneous processors that offer specific functionalities reflecting the need of the expected application domain, for example FPGAs or *Application Specific Instruction-set Processors* (*ASIPs*) in addition to general purpose processors. MPSoCs embody an important and distinct branch of multiprocessors, especially in applications that require a flexible computing structure. MPSoCs are widely used in networking, communications, signal processing, and multimedia among other applications [WJM08]. Hardware examples are the commercial multimedia platforms *NXP Nexperia* and *ST Nomadik* that use MPSoCs, and the nine-core *Cell Broadband Engine Architecture* developed by Sony, Toshiba, and IBM that is used in the game console *PS3* [KBDV08].

Distributed Memory Multi-Processor A distributed memory system is a cluster of multiple processors in which each processor has its own private memory. There is no global memory and changes to local memories have no effect on the memory of other processors. When a processor needs access to data in another processor the data must be explicitly communicated. Hence, distributed memory systems require a communication network to connect inter-processor memory. The interconnection can be organized with point-to-point links or separate hardware can provide a switching network using some standard network protocol, for example Ethernet, or using *Dual-ported RAM* (*DPRAM*). Because the latency for obtaining data in memories of other processors is higher than it is the case for accessing data in own memory, the distributed memory architecture is a NUMA system.

Since memory addresses in one processor do not map to another processor, there is no concept of global address space across all processors. There-

fore, such distributed memory systems are also sometimes called *NORMA*, for *No Remote Memory Access*. As a result, the advantage of distributed memory is that it excludes race conditions[6]. In contrast to SMPs, the processors of the distributed memory system do not have to be identical. An early project that showed the viability of the concept was the *Stone Supercomputer* with 133 processors. It included *Intel 80486* and *Pentium*-based machines and a few *DEC Alpha* workstations [HHS01]. The world's fastest computing system in 2012, which is published via the TOP500 organization's semiannual list of the 500 fastest supercomputers [Kra12], was a distributed memory cluster, the *Cray Titan* supercomputer [BMPW12] with 18,688 *AMD Opteron* 16-core CPUs and 18,688 *Nvidia Tesla GPUs*. As published via the TOP500 list in November 2013, the *Tianhe-2* distributed memory supercomputer with 260,000 *Intel XEON* 12-core CPUs is the fastest in the world at the present moment.

With this thesis, we support the mapping of concurrent applications to parallel architectures that realize any of the architectural models mentioned in this section. The next section gives background on the challenge of mapping the computational tasks of applications to these architectures.

2.6 Task Mapping

Mapping concurrent applications to parallel architectures refers to the problem of scheduling the parallelly executable tasks of the application and allocating the tasks to a limited number of PEs. Mandelli et al. [MAOM11] propose a taxonomy for task mapping techniques. The authors classify task mapping according to the following three criteria:

- **Point in Time of Mapping** The point in time of mapping can be either at compile time, called *static* or *offline*, or at run-time, called *dynamic* or *online*. Static mapping may employ complex heuristics using thorough information about the system to better explore optimization potential. This information includes the topology of the task graph, the predicted communication between tasks, and the availability of PEs. However, in general there is no information about the run-time behavior of the application available at compile time without a previous profiling step. Dynamic mapping require simple and fast heuristics because the time for mapping at run-time interferes with the total execution time of the application.

  Mandelli et al. [MAOM11] additionally differentiate dynamic mapping techniques regarding whether there is one entity responsible at run-time for managing the mapping or more than one and whether the mapping reserves available resources of the target architecture or not. Without

[6] *Race conditions* arise in software when parallel computational tasks depend on some shared state. Operations upon shared states are critical sections that must be mutually exclusive. Violating this condition opens up the possibility of corrupting the shared state, thus leading to a race hazard.

reservation of resources, the mapping heuristics maps one or more initial tasks of the applications (those without dependencies to other tasks) and the remaining tasks when they are required. This approach may start applications faster, but some tasks may wait for available resources if the system usage is high. With reservation of resources, the mapping heuristics verify if there are enough resources of the target architecture available before the mapping.

- **Number of Tasks Mapped per PE** The task mapping technique is called *single-task* when at most one task is allocated to each PE. When more than one tasks can be allocated to each PE, mapping is called *multi-task*. The latter technique requires a partitioning approach to define a group of tasks to be mapped to the same PE. Which approach is more suitable depends on the granularity of PEs. For example, a multi-core processor with multiple CPUs can be defined as one single PE or each core can be defined as a distinct PE. For the former case, single-task mapping is more appropriate, and for the latter case, a multi-task mapping technique is more appropriate.
- **Target Architecture** The mapping can be classified according to the architectural model of the target architecture. It is a *homogeneous* mapping technique when all PEs are identical and a *heterogeneous* mapping technique when the target architecture is composed of different PEs. For heterogeneous mapping, the additional subproblem of determining which PE can execute each task most efficiently has to be necessarily solved.

In general, the problem of optimal task mapping to a limited number of PEs is computationally complex since the corresponding decision problem for partitioning is NP-complete [Kar72] and, furthermore, it is hard to approximate [Zuc96, KP01]. Communication-aware allocation of tasks to PEs and multi-task mapping additionally requires partitioning tasks into groups of tasks according to their communication characteristics or according to the mapping of groups of tasks to PEs, respectively. Due to its computational complexity, task mapping requires heuristics to obtain near-optimal solutions for real-world applications in an acceptable timeframe. These heuristics demand the run-time behavior of tasks to be known. To facilitate an automatic mapping, this information has to be given by automatic analyses.

With this thesis, we aim at improving the mapping of concurrent applications to parallel architectures. According to the classification of Mandelli et al. [MAOM11], we directly support the static, single-task, and heterogeneous mapping technique. Since our approach both automatically analyzes the run-time behavior of tasks and determines the mapping at compile time, it is a static technique. However, our approach can also be used to improve dynamic mapping techniques as we demonstrate with our instantiated framework for intelligent mapping of MPI programs to processor networks (see Chapter 7). With our predictions for the execution times and the communication amount, a run-time scheduler is able to allocate tasks communication-aware and power-efficient. If the information was not given (e. g., via our automatic predictions or manual annotations) and the tasks were not periodic, the run-time scheduler

would not have the relevant information and can do nothing except performing workload balancing. Our approach defines each computational resource of the target architecture as a distinct PE. Hence, single-task mapping is more suitable. If we had defined a multi-core processor as one PE, we also would have supported multi-task mapping. Since we do not constrain the PEs of the target architecture to be equal, we support heterogeneous mapping, which is the major challenge compared to homogeneous mapping.

2.7 Summary

In this chapter, we have presented the relevant background to our compiler framework for ML based mapping of concurrent applications to parallel architectures. We have introduced both the essential components of a compiler and program analyses, which are vital to compiler optimizations. We have surveyed ML techniques used by our approach for automatically incorporating predictions of expectable run-time behavior into the compilation flow, namely supervised classification learning and regression modeling. To assess the resulting predictors, we have characterized quality measures of both the ML model and the precision of predictions. With this thesis, we aim at improving the mapping of arbitrary applications to any target architectures. Therefore, we have reviewed supported parallel programming models used to express the applications and have given introduction to the main classes of parallel architectures targeted by our approach. Finally, we have described the challenge of task mapping.

3 Related Work

Within this thesis, we propose a compiler framework for task mapping to parallel architectures, which is, among other things, aware of the communication between parallelly executable tasks. To achieve this, we use machine learned loop iteration counts together with machine learned recursion frequencies to estimate the communication overhead at run-time if communication arises within loops or recursive functions, respectively. To improve task mapping, predictions for the execution time of tasks are also required. Again, our approach is to use *Machine Learning* (*ML*) techniques to derive useful predictions for expectable execution times. In this chapter, we discuss related work and we compare it to our approach with respect to the use of ML techniques in compilers, the prediction of necessary run-time behavior, and the challenge of task mapping.

3.1 Machine Learning in Compilers

The application of ML techniques in compiler frameworks has become an increasingly popular research area. The key advantage of learning techniques is their ability to find relevant information in a high-dimensional space, thus helping us to understand and control complex systems.

Dubach et al. [DCF+07] use ML to predict which optimization sequences of compiler optimization passes may be advantageous for which kinds of sequential programs. They use combinations of 54 different program transformations restricted to sequences of up to a length of 20. As code features, characteristics of computation operations, memory access computations, and control-flow operations are used. As a result, applying predicted optimization sequences finds about 80% of available performance improvement. The approach of Fursin et al. [FMT+08] targets the automatic selection of good optimization passes and heuristics that are associated to the best combination of compiler flags for the compilation of sequential programs. During the training phase, information about the program structure is gathered using 55 static program features and the behavior of programs when compiled under different optimization settings is recorded. After training, on encountering a new program the optimiza-

tions to be applied are selected. The authors have used 19 benchmarks of the MiBench benchmark suite [GRE+01] to evaluate their approach. As a result, automatically predicted optimization passes using ML improves performance by 11% on average. Dubach et al. [DJB+09] extend this approach to develop a portable optimizing compiler. They use a description of the target architecture together with hardware counters from a single execution of the program to predict good compiler optimization passes across different generations of the target architecture. Using this approach achieves 16% speedup over the highest default compiler optimization on average. To compare multiple compiled versions with different compiler flags, the approaches above require the programs to be executed multiple times. However, the computation time to profile all these different versions of the training benchmarks can be on the order of weeks or months [FMT+08]. The work of Leather et al. [LOW09] addresses this issue by automatically selecting the number of times to execute each program version with the use of ML techniques to minimize the number of executions required. They determine a subset of the optimization settings and program versions that perform better, dropping poorly performing versions early and finishing when sufficient data has been gathered for a decision. They show that their technique is able to correctly determine the best optimization settings with between 76% and 89% fewer execution runs than needed by a brute force constant sampling size approach.

Lokuciejewski et al. [LGMM09] focus on using machine learned heuristics for *Function Inlining* in order to reduce the *worst-case execution time* (*WCET*), which is an upper bound on the maximal possible execution time of a program. Their supervised learning algorithm yields a classifier that decides whether inlining of a particular function promises a WCET reduction and should thus be applied. This binary classification is based on in total 22 static code properties and features obtained from a static WCET analysis, such as the number of instructions to be executed of the caller and the callee, whether the considered call to the function is invoked once or contained in a loop, features that indicate register pressure, and the WCETs of the caller/callee as well as the execution frequency of the call derived from the WCET analysis. Lokuciejewski et al. [LSMM10] build upon this approach by evaluating other learning algorithms. They present an algorithm that automatically selects the best ML model and its parameters[1]. The optimization of the model parameters is based on evolutionary algorithms. The goal of their work is to minimize the WCET with a learned predictor that decides whether *Loop Invariant Code Motion* should be applied. To that end, they use code features describing characteristics of single instructions, basic blocks, loops, and functions. The aim of Lokuciejewski et al. [LPF+11] is to automatically find optimization sequences of compiler optimizations with conflicting goals. They consider in total 28 standard code optimizations like *Constant Folding*, *Common Subexpression Elimination*, *Constant Propagation*, and *Dead Code Elimination*. Instead of the reduction of the WCET as single objective function, the combination of WCET with *average-case execution time* (*ACET*) and WCET with the result-

[1] As introduced in Section 2.3, learning algorithms have special model-parameters, such as the k for the k-NN regression model.

ing code size is used as multi-objective functions. Their approach approximates Pareto optimal solutions for this problem via multi-objective evolutionary algorithms.

All ML based approaches mentioned in this section aim at improving the performance of *sequential* programs by automatically adapting certain compiler optimization passes. In contrast, we use ML techniques to predict certain program behavior within our proposed framework for guiding the automatic mapping of *parallel* applications in order to improve their performance, which has never been considered before.

3.2 Predicting Loop Behavior

To statically determine the communication overhead of parallel tasks at run-time, predictions for loop iteration counts are required. Though pure static analyses also can estimate loop iteration counts, the estimates are imprecise because iteration counts of loops are rarely predictable using static techniques. Wu et al. [WBD04] have shown that even profile information initially gathered at run-time is inadequate for predicting the loop behavior (and statically not even this information is given). The authors propose a profiling algorithm at run-time with a low overhead that can be used by dynamic binary translators to enable advanced loop optimizations. This algorithm records run-time information of loops during execution of the program, groups ranges of the loop iteration counts into three classes, and reports a change of the loop behavior when during a phase of the execution a change of the classified iteration count occurs. In contrast, we aim at *statically* predicting loop iteration counts in advance without the need for having (initial) run-time information of the current application, which is the major challenge. In the following, we discuss other existing static approaches to obtain necessary information about the loop behavior.

Machine learned heuristics for the loop behavior at run-time often outperform hand-crafted models (cf. [MBQ02]). As an example application, the approach of Monsifrot et al. [MBQ02] learns to decide if *Loop Unrolling* is beneficial or not. The used static code features are the number of statements, arithmetical operations, array references, array element reuses, if-statements, and the minimum of loop iterations. Stephenson and Amarasinghe [SA05] use supervised classification learning based on static code features to predict the best loop unroll factors. They use 38 features in total, such as the number of operations and branches in the loop body, the estimated latency of the critical path in the loop, and the programming language the loop is written in. The goal of Long et al. [LFF07] is to determine the best parallel scheme for workload allocation, i. e., how many threads should be used for loop parallelization. In the training phase, loop features such as the loop depth, loop size, number of arrays used, statements in loop body, and array references are captured. A new category is created in the database if none of the existing ones has a similar workload. Then, the loop description is stored within the category,

together with different parallel schemes and the corresponding performance improvement. When a new loop is encountered after the training phase, its features are captured and the loop is classified. The most similar loop category within the database is identified and the best scheme is selected. Though all ML based approaches above aim at predicting loop behavior they do not predict loop iteration counts with ML techniques as we do, which has not been considered before to the best of the author's knowledge.

Closer to our approach is the work of Buse and Weimer [BW09] for statically estimating the path execution frequency using binomial logistic regression. Their proposed learning technique for programs written in object-oriented languages statically predicts execution frequencies of paths within a class. It is based on static code features that indicate whether a path is intended to execute less (like raising an exception after an assertion failure) and that measure the percentage of possible occurrences of language constructs along the path (like percentage of invoked method statements along a path). However, the outcome of their learning algorithm is only a binary judgment partitioning the paths into low frequency and high frequency. In contrast, we provide a more fine grained classification. Another non-trivial heuristics for predicting loop iteration counts without profiling is the approach of Wu and Larus [WL94] for statically predicting execution frequencies of *basic blocks* (*BBs*) within the *Control Flow Graph* (*CFG*) of functions. It is based on local predictions of the probability whether a branch in the CFG is taken or not and derives the execution frequency via the *Call Graph* (*CG*) of the program. However, our experiments show (see Subsection 7.2.1) that our approach is more precise as is demonstrated by our lower mean absolute error and the higher correlation.

There exists related work for statically predicting loop iteration counts that support WCET analysis for real-time systems. These approaches yield, whenever possible, a safe upper bound on the number of iterations a loop may execute, valid for all possible input data of the program. Healy et al. [HSR+00] propose a semi-automatic analysis for assembly programs that tries to determine the minimum and maximum number of iterations of loops with multiple exits. It identifies the conditional branches within a loop that can affect the number of loop iterations and tries to calculate the range of possible iterations based on the condition of these branches. The calculation succeeds when all of the following requirements on the condition hold. The control variable on which the branch depends must be a *basic induction variable*, i. e., a variable that is solely decremented or incremented by a constant integer, and the constant change of this variable must be on every complete loop iteration. Furthermore, the initial value of this variable when the loop is entered must be an integer constant. The condition must be a relational operation between the variable and a value, where the value being compared to must also be an integer constant. If one of the requirements above is not satisfied, i. e., the loop is dependent on unknown values of variables, the user is asked to provide bounds for these variables manually. The same constraints apply for the approach of Corti et al. [CBG01] that tries to obtain tight predictions for the WCET, among others, via static analyses of loop bounds. The authors state that a "simple" loop termination analysis, which is not specified, is performed to de-

rive these (unsafe) predictions. They mention that it cannot handle multiple loop exit conditions and this single exit condition must not depend on input values, i. e., the value to be compared with must be a constant. If loop bounds cannot be statically determined, they require the maximal loop iteration count to be annotated by the user. The authors report for experiments with three applications that 19 loops (from an undisclosed total number of loops) cannot be analyzed. Cullmann and Martin [CM07] use an interprocedural *Data-flow Analysis* (*DFA*) on assembly level to calculate the interval of the possible loop iteration count for safe WCET analysis. The DFA is able to handle loops with multiple exits and multiple modifications of the loop counter per iteration including modifications in procedures called from the loop. Based on the result of the DFA, an equation system is built and solved to get the concrete loop bound. Unlike the work of Healy et al. [HSR+00] and Corti et al. [CBG01], this approach does not require that a change of the basic induction variable must be on every complete loop iteration. However, their analysis also fails to produce a result if variables on which the loop iteration depends are not initialized with a constant value. Furthermore, the analysis can only handle modifications to these variables that add a constant integer (interval). That is, other modifications like non-constant addition or any kind of multiplication are not supported. In contrast to the approaches for WCET analysis mentioned above, our ML based approach is fully automatic in all cases without having these requirements. During our experiments, we have found that only 1057 out of 24880 loops which were called at run-time of the program have statically determinable loop bounds via constant initialized loop induction variables. For these loops, our analysis also yields the exact loop iteration count. Otherwise, our analysis yields a precise (but not safe) approximation.

For WCET analysis, several fully automatic approaches to predict loop iteration counts based on *Abstract Interpretation* (*AI*) were proposed. AI is a theory of sound approximation of the semantics of the program by mapping the concrete semantics of the program to an abstract domain with elements representing sets of abstract states. The idea of using AI is to gain information about possible program behavior via abstract execution of the program without performing all the calculations. Given an initial abstract state that specifies possible constraints on input variables of the program, a set of transfer functions that map abstract states to abstract states are used to build an equation system that relates the abstract states for different program points. The set of equations is solved via *least fixed-point iteration*. Ermedahl and Gustafsson [EG97] use an AI based DFA with an abstract domain where intervals can be assigned to variables. Using this domain, the abstract execution of the program yields for each point of execution an environment which holds all combinations of variable values that are possible. Then, given the initial intervals of possible values for all variables, a loop is "rolled out" until it cannot execute again or a maximal time budget that must be defined is reached (because, e. g., the abstract execution will not terminate if loop counter variables are unbounded or if the program does not terminate). However, their used abstract domain of *split integer intervals* has the drawback that it is not possible to express relations between variables. Furthermore, AI might slow down the analysis such that it becomes impractical. In particular, this can be ob-

served for the analysis of program loops with high iteration counts for which each loop iteration is separately simulated [LCFM09]. For example, the time needed for AI based loop bound analysis is reported in the work of Gustafsson et al. [GESL06] that extends [EG97] with support for nested loops. They have used 14 benchmarks from the Mälardalen WCET Benchmark Suite [Gus07] with up to 4253 lines of code and 12 contained loops. To speed up the analysis, they use *Program Slicing* to restrict the abstract execution to only those program parts that may affect the program flow, but their loop bound analysis still needs more than 36 seconds analysis time per benchmark at maximum (with an average of 5.7 seconds). The authors state that their AI based execution has a potentially bad worst-case complexity and no guaranteed termination [ESG+07].

To further reduce the time needed for AI, Ermedahl et al. [ESG+07] use additionally *Loop Invariant Analysis* to identify variables not having their values changed in loops and to remove these variables from the corresponding loop bound calculations. Since AI is input dependent, bounds on input values can be manually specified. Their abstract domain can, in addition to integer variables, also handle other basic data types, such as characters, floating-point types, aggregate data structures, and pointers. However, their approach is less general because it cannot compute bounds for all types of loops. If the variable is or contains a floating-point value then they consider the number of concrete values to be either zero, one, or infinite. If any variable in the abstract state is represented by an infinite set of concrete values, then the loop bound cannot be derived. Furthermore, as stated by the authors, their analysis cannot handle recursive code and it often fails to derive loop bounds for more complex loops or loops containing floating-point counter variables. Using 28 programs of the WCET Benchmark Suite with up to 4253 lines of code and 30 contained loops, their analysis derives for 63% of the loops an upper bound, with a maximum analysis time of more than four minutes per benchmark (with an average of about 18 seconds).

The approach of Lokuciejewski et al. [LCFM09] for a static loop analysis based on AI exploits the mathematical concepts of polytopes to reduce the analysis time by reducing the cases where the time consuming AI technique must be used. This is accompanied by interprocedural *Program Slicing* and a context-sensitive alias analysis to support pointers. They propose a non-iterative static loop evaluation based on polytopes which determines loop iteration counts and variable values with a purely static analysis by iterating through the loop body exactly once. It is well-suited for WCET for safety-critical systems where the result must be a *safe* approximation. However, a polytope based solver has certain restrictions. The loop exit condition must be based on a constant, a non-modified variable within the loop, or a single variable. Furthermore, the conditions that influence the loop execution must be affine, i. e., it must be a linear expression of the form $Ax + b$ where A, b are constants, and all assignments in the loop body must be transformable into the form `=`, `+=`, or `-=` to ensure that the variable is increased in each loop iteration by a constant value. Moreover, aggregate types and pointers are not supported by the polytope based analysis. If a loop satisfies all these requirements, they

can transform the loop into an integer polytope model and the non-iterative evaluation can be performed. If one of these requirements is not satisfied, the classical analysis based on AI is applied. For their experiments, they use 96 programs from different benchmark suites that contain 707 loops in total, but they exclude some complex benchmarks which cannot be handled. For 1% of these 707 loops their approach fails to analyze the loop iteration count at all and only 21% of the 707 loops could be analyzed with the polytope-based non-iterative loop evaluation. The remaining 79% must be analyzed using the time consuming AI technique together with *Program Slicing*. They do not report analysis times for all benchmarks, but they state that smaller benchmarks require a few seconds and larger benchmarks less than four minutes on average.

Recently, Knoop et al. [KKZ12] aim at automatically inferring loop iteration bounds for a restricted class of loops to improve WCET analysis. The restricted class they consider encompasses loops where all updates over program variables must be linear expressions. They distinguish loops with a single control flow path in the CFG of their body, called *simple* loops, from loops with multiple paths (arising from conditional jumps in the CFG), called *multi-path* loops. For the former they model the iteration update as a recurrence equation over a new variable denoting the loop counter. This equation is solved and the closed form for the loop induction variable is derived, which is used to compute the smallest iteration where the loop invariant is violated. Some loops of the latter category are transformed into simple loops by refining the control flow, which can be done in two cases. First, an expression that causes the termination of the loop iteration (e. g., a `break` statement) occurs in a path of the CFG that is executed depending on a condition. These loop exits are completely ignored. This safely over-approximates the number of iterations because control flow paths that exit loops are assumed to be not taken. In contrast to this work, loop exits are not ignored by our approach and are instead a feature as input to our learning algorithm because the more exit branches a loop contains, the greater the likelihood that the loop iteration is actually terminated. This statistical assumption holds since each exit has a certain branch probability, which sum up to this likelihood. Hence, loop exits should not be ignored. Second, their method can handle loops where a linear update of an induction variable occurs in each of both paths of a conditional branch in the CFG. In this case, its condition is ignored and the minimal value by which the induction variable can be increased is computed and used as update. Other multi-path loops than discussed above cannot be handled by their approach. The authors evaluate their approach on four benchmark suites containing 338 loops in total and derived loop bounds for 253 loops. Out of these 253 loops, 243 were simple loops and only 10 of them were multi-path loops.

In contrast to the approaches for safe WCET estimates discussed above, our analysis which solely has to extract static code features during compilation always terminates and predicts loop iteration counts for all loops without any restriction, though yielding unsafe estimates. Furthermore, it is highly scalable with a negligible compile time overhead. In addition to using our predictions of

the loop iteration count for estimating communication overhead between tasks, our approach can also be used to guide other loop related optimizations. For example, the approach of Lam et al. [LCLL09] can use our result instead of using an estimated loop iteration count that otherwise must be provided by the programmer. Our predictions can further be used to guide *profile-based optimizations* like the approaches of Hotta et al. [HSK+06] or Feng et al. [FDM+08] for optimizing the power performance or memory accesses of applications, respectively. Our predictions can also guide approaches to predict *hot spots* of programs, which are code fractions where most of the execution time is spend. Program execution tends to spend most of the time in a small fraction of code, a characteristic known as the "90-10 rule" – 90% of the execution time comes from 10% of the code. About 85% of those regions are inner loops, while the remaining 15% are functions [SNV+03]. The study of Villarreal et al. [VLCV01] also shows that some of the programs spend over 90% of their execution time in loops. The average across the examples used in their study is roughly 70%. They additionally observe that many loops iterate only once or just a few times, which is also confirmed by our observation. Many heuristics have been proposed for hot spot detection at run-time, e.g., used in just-in-time compilers or in JAVA virtual machines [KGSC01, ARPS06, LMK08, CSACR09]. But even run-time estimation techniques oversimplify either loop iteration counts or loop/method sizes in order to reduce the estimation overhead [LMK08]. This can lead to imprecise hot spot detection. Common to these approaches above is the use of profile information gathered at run-time. Hence, to be used in a static compiler, they have the penalty to additionally requiring to execute and to re-compile the program after its execution for enabling hot spot optimizations. In contrast to hot spot detection at run-time, our approach statically predicts loop iteration counts without the need for profiling at compile time, which is the major challenge.

3.3 Predicting Recursion Frequency of Functions

With this thesis we aim at, among other things, reducing the communication overhead between parallelly executable tasks at run-time of a concurrent application. Since estimating the overhead involves estimates for the communication amount, we use predictions of the recursion frequency of functions[2] in case of communication arises within recursive functions. We do not use the recursion depth because it would be an underestimate of the whole communication amount if the function calls itself more than once in one recursive step.

To the best of our knowledge, no work was yet done to predict the recursion frequency itself and there are only two approaches that predict the recursion depth. These two known approaches handle only very special cases. The ap-

[2] *Recursion frequency* refers to the number of self calls of a function resulting from one external call to this function at run-time, in contrast to *recursion depth*, which refers to the number of levels of activation of a function which exist during the deepest call of the function.

proach of Stitt and Villarreal [SV08] tries to analyze the maximum recursion depth of functions for inlining the corresponding calls until the detected recursion depth is reached to enable hardware synthesis. They analyze the stopping conditions that guarantee that a recursive call will not execute. However, for all variables that are used in a stopping condition or define other variables used in a stopping condition they require a constant initialization and only monotonic updates. Hence, it can only be applied for rare algorithms. The second approach of Kirschenhofer and Prodinger [KP84] handles only traversal algorithms over *planted planar trees* with known size. In that case, the depth can simply be derived from the height of the tree. Therefore, their approach is more strongly limited to special algorithms. In contrast to both approaches before, our proposed solution for predicting the recursion frequency is not limited to special cases. Additionally, our solution can extend both approaches. If their analyses knows the result to be exact, a compiler optimization can take their result as heuristic, otherwise our predictors can be consulted.

Apart from using learned predictions of recursion frequencies within our framework for ML based mapping of concurrent applications to parallel architectures, our predictors can also enhance static WCET prediction in all cases where the result has not to be a safe approximation. For example, the approach of Corti et al. [CBG01], which we discussed in the previous section, requires recursive functions to be rewritten as a loop. Though it is in principle possible to rewrite *tail recursive* functions as a loop, the stopping condition of recursion becomes a loop bound that must be predicted. During experiments with three applications to evaluate their approach, nine recursive functions must be manually transformed to a loop. Using our predictions for recursion frequencies as estimate, manually rewriting of code would be no longer necessary. Likewise, our approach can eliminate the need for user annotations of recursion frequencies, whenever the value to be annotated is used by static compiler analyses that do not rely on the correctness of it. For example, the approach of Ding and Li [DL03] requires annotations of unknown loop iteration counts and recursion depths to obtain precise estimates for interprocedural data communication. Both information can also be automatically given by our learned predictors for the loop iteration counts and recursion frequencies.

Close to our approach for learning recursion frequencies is the work of Buse and Weimer [BW09] for statically estimating the path execution frequency as discussed in the previous section. However, they consider execution frequencies of paths instead of number of recursive calls to functions as we do, and the outcome of their learning algorithm is a binary judgment partitioning the paths into low frequency and high frequency. Instead, we have used a more fine grained classification. The approach of Wu and Larus [WL94] for predicting execution frequencies of BBs in the CFG, which we discussed in the previous section, can also be used to predict the recursion frequency of functions. Again, our approach is more precise as is demonstrated by our lower mean absolute error and the higher correlation (see our experimental results in Subsection 7.2.2).

3.4 Predicting Execution Time

For scheduling parallelly executable tasks of an application and for their allocation to PEs of the target architecture during task mapping, execution times of tasks must be predicted. Although this can be done via profiling (see Subsection 2.2.2), it has not been widely adopted due to the tedious dual-compilation model, the difficulties in generating representative input data sets, and the high run-time overhead of profile collection [CVH+13]. Therefore, many approaches have been proposed to determine this information based on previous observations. In this section, we discuss related work in this area and compare our approach to that.

Iverson et al. [IOF96] have introduced a scheme for predicting execution times of tasks on a heterogeneous target architecture. Their scheme derives these predictions at run-time of applications just before a task is executed. It is based on a *k-Nearest Neighbor* (*k-NN*) estimator with a regression function that takes as argument a scalar parameter x. Then, given a new parameter a, an estimate is constructed from the k observations having x values closest to a. The authors mention that their scheme can be extended to estimate the execution time using a vector of parameters. However, they do not specify the information that the parameter x (or the vector of parameters, respectively) holds. That is, no facts are given how knowledge from a task, e. g., properties of the source code, is condensed to parameters. Instead, they use generated random values as parameter x during their simulations to evaluate the proposed method and they choose a particular function m instead of a regression function, which they also do not mention. To simulate the execution time of tasks, it is computed to be the value of $m(x)$ plus some zero mean random error. Hence, in contrast to our work, neither are the features defined nor have been performed practical experiments that show the applicability of their approach with real data.

The approach of Iverson et al. [IOP99] builds upon this work. Their proposed method predicts the execution time as a function of a vector of parameters that holds input and performance data. Input data is the argument an application is invoked with. Performance data refers to varying execution times of the same application on several target architectures. For their experiments, they consider two algorithms for which the execution time is estimated, an algorithm that attempts to determine if a given number n is prime through trial division and a Cholesky matrix decomposition algorithm. For the former case, the magnitude of the value of n is used as input parameter and for the latter, the size of the matrix is used. That is, predicting execution times for new applications with variant arguments requires to adapt the parameter vector, which is a major drawback because an expert has to do this manually. As performance data to characterize the differences in executions times, they have used 10 benchmarks of the Byte benchmark suite, which were executed on 16 different architectures. Another drawback of their execution time estimation algorithm is the source of an initial set of observations. Every time a given application is run on a machine in the target system, a new observation is

added to the set of previous observations. Since the execution time estimate is a function of the set of previous observations, at least one initial observation is required when a new application is introduced into the system. Thus, an unseen application must be executed on several selected machines in order to obtain a few initial estimates. In contrast, our approach for predicting execution times is automatically applicable without the need for manually adapting the parameter vector and without requiring to execute unseen programs several times. Furthermore, our approach is applicable at compile time because it does not depend on run-time arguments of applications.

The work of Corti et al. [CBG01], which we have discussed in Section 3.2, aims at determining an approximation of the WCET for programs written in object-oriented languages. To obtain a prediction, the program is first compiled and the resulting object code must run on the target platform with some training input. The execution of this program is monitored in the actual system with the same set of processes that will be present in the final system. Thereafter, their approach uses run-time statistics of executed instructions from profiling runs as well as the source code to approximate the WCET. First, they compute the longest path in the *data flow graph* (*DFG*) of the program, where nodes are the BBs of the CFG. When each loop in this DFG is bounded (see Section 3.2) and hence the number of iterations of each BB is known, the number of cycles needed to execute each BB is computed. To that end, they use gathered run-time information of the hardware performance monitor (e. g., cache misses, idle and busy cycles of functional units of the CPU, and pipeline stalls) to approximate it. Then, edges of the DFG are weighted with the number of cycles needed to execute the corresponding source BB. Finally, these cycles are multiplied with the number of times a BB is executed to derive the needed cycles for execution of the longest path. A major drawback of their technique compared to our approach is the requirement to profile each program several times with typical input data. Additional restrictions on their technique that are not imposed by our approach are required manual annotations of maximal values for those loops and method invocations for which a compiler cannot determine the iteration count or call chain. Furthermore, the necessary cycle counts for reading input data or writing output data must be annotated by an application expert. Moreover, they prohibit the use of the `new` construct in programs, since this operation is not bounded in time or space. In contrast, our approach permits all constructs of the programming language.

Giusto et al. [GMH01] have proposed a method for estimating execution times that also permits all constructs of a programming language, in this case C code, which can be used to speed up simulation time. Their method first translates actually executed instructions of a program into a set of 25 *virtual instructions*. Then, an estimate for the execution time in terms of processor cycles is given by a predictor equation $Cycles = K + \sum_i P_i * N_i$ where N_i is equal to the number of executed virtual instructions of type i, P_i is the cycle count of instruction i, and K is the intercept. Since the number of virtual instructions to be executed must be known, their technique is only applicable during simulation when this information is present. For solving the equa-

tion, P_i can be determined using linear regression and expert knowledge that the authors called "users intuition". This intuition must be used to remove certain instructions i from the predictor equation to obtain better results. They have used 35 control-oriented software tasks drawn from a real automotive application to establish a predictor and have evaluated its precision using six Esterel benchmarks. The established predictors have performed poorly, overestimating the cycle counts by 87% to 184%. The authors argue that only tasks that are similar to these used to establish the predictor can be predicted with confidence. If not, their conclusion is that a set of applications with similar code should be used to establish a new predictor. To measure similarity, they propose to use the ratio of the number of `IF`-instructions to the total cycle count, which determines the *control-dominance* of a task. In contrast, our approach is automatically applicable without the need for knowing the number of instructions to be executed. Furthermore, if the number of instructions to be executed is known, our approach performs better as our experimental results show (see Subsection 7.2.3).

Close to the work discussed above, the method of Bontempi and Kruijtzer [BK02] is also based on regression modeling with a set of virtual instructions and the number of times each instruction is executed as features. Their method extends linear to nonlinear regression techniques, called *lazy learning* and additionally includes architectural parameters in the feature vector to predict execution cycles of applications. They use the number of memory wait cycles and the ratio between CPU clock and memory clock as architectural parameters of a virtual processor. To evaluate their approach, they use six small algorithms, e. g., for sorting, mathematical computation, and inverse discrete cosine transformation, with 20 to 370 lines of code, each one executed 15 times with different input data sets. Additionally, they use a variable length decoder algorithm of the MPEG2 decoder consisting of about 5000 lines of code, which they consider as a more realistic setting for testing the capability of their method. This algorithm is executed with two input files containing 16 pictures in total and a measure of the execution cycles is taken at the end of each picture. For their experiments, they report a percentage error between 8.8% and 17%. However, there is no evidence how their approach will generalize to a greater test set with more "realistic" programs. Compared with this method, we have used 391 programs of 30 benchmark suites from different real-world application domains to evaluate the accuracy of our predictions. Furthermore, our learned predictor for the execution time is statically applicable without having information about arguments given to programs and the number of times instructions are executed at run-time, which is the major advantage.

Oyamada et al. [OZW04] propose to use *neural networks* for estimating execution cycles. They classify instructions into five classes that capture possible behavior: integer calculations, floating-point calculations, backward and forward branches in the CFG, and memory accesses. As with the approaches before, their method requires the number of times of executions of these classified instructions to be known. But instead of using numbers obtained from profiling of virtual instructions, the application is compiled for the instruction set of the target processor, which requires compiling the application and pro-

filing it for each different processor to be evaluated. This dominates the time consumed by the utilization phase of their method for obtaining predictions, since it takes about five minutes for a given application as reported by the authors. In contrast, our technique does not require to profile each application since our learned predictor for the execution time is statically applicable. Their approach yields the same accuracy compared to the work of Bontempi and Kruijtzer [BK02] but they have used a more heterogeneous benchmark set consisting of 42 programs for their experiments. To improve the precision, they propose to automatically classify applications to distinguish control-flow from data-flow oriented application domains, which yields two classes. The classification is based on the computation of a *CFG weight*, based on the number of arcs connecting the basic blocks. The benchmark set is thus divided in a first set with 20 control-flow oriented applications and a second set with 21 data-flow oriented applications. Using domain-specific estimators instead of a generic estimator for all programs decreases the mean percentage error from 7.9% to 6.4%. However, they have kept only 16 benchmarks from the first set and removed four benchmarks with floating-point instructions that could harm the generalization of the neural network.

The approach of Smith et al. [SFT04] aims at predicting execution times of parallel workloads based upon observed execution times of similar workloads. They use search techniques to determine those characteristics that yield the best definition of similarity for the purpose of making predictions. Based on this, *templates* are defined that identify a set of *categories* to which workloads can be assigned. For each architecture, the initial set of categories is empty. As characteristics of workloads, they use features such as the name of the queue to which a workload can be assigned on supercomputers, the name of the workload, the name of the user that starts the workload, arguments to the workload, and the number of computing nodes of the supercomputer that are requested for executing the workload. For example, a template consisting of queue and user name specifies that workloads are to be partitioned by the user and queue. This template generates categories, each one being an element of the cross product of all queues and users of the supercomputer. At the time an application begins to execute, the templates are first applied to the characteristics of the application to identify the set of categories into which the application may fall. Then, all categories within this set are eliminated that are not in the initial set (which may be empty) or that cannot provide a valid prediction (i. e., do not have enough previous observations). If the remaining set is not empty, run-time estimates are computed for each category and the estimate with the smallest confidence interval is selected as prediction. If it is empty, no prediction can be made. That is, their approach needs initial observations of workloads with similar features such as the same user or workload name, which are identified by the authors as most important characteristics to use when deciding whether workloads are similar. Our approach, on the contrary, is always applicable independent of user names or names of executables, and does not require to know arguments the executable is invoked with. The evaluation of their approach using four workloads traces recorded from supercomputers centers shows prediction errors between 29% and 54% of

mean execution times, which is worse than reported errors from the work of Oyamada et al. [OZW04] that we have discussed before.

Singh et al. [SIM+07] also aim at predicting parallel application performance via ML. They use *multilayer neural networks* to predict the execution times of two concurrent applications, SMG 2000, a semicoarsening multigrid solver, and HPL, an open-source implementation of LINPACK. Their basic idea is to establish an application-specific model where the prediction solely depends on the characteristics of the input data to this application at run-time without considering characteristics of the application itself (as, for instance, executed instructions). Then, having learned a neural network for a particular application, a prediction for this application is derived based on its current input data. As input characteristics for SMG 2000, they use six parameters of the parallel solver that describe both the shape of the workload per processor and the logical processor topology. For HPL, they use its five parameters that are most significant for configuring the solver and vary the processor grid topology. For both applications they choose the default values for all other parameters. Varying the assignment of possible values to these parameters leads to configuration spaces with 3358 to 6170 points. To evaluate the accuracy of the model, each configuration space is separated into two disjoint sets for training the neural network and for testing the precision of the prediction. The sample training set is iteratively incremented by 50 points at each round and the precision of predictions is calculated for 1000 randomly sampled points of the remaining configurations. Their experimental results show that at least 5% of the configuration space must be used for training to obtain prediction errors below 17%. Using training sets with 30% of the configuration space can predict the execution time with 2% to 7% error, which means that an application had to be executed about 2000 times for training to obtain these error rates. Since their performance model is application specific, distinct neural networks must be learned for each unseen application. That is, a new application must be executed and profiled up to 2000 times before a precise prediction can be obtained, which is a major disadvantage compared to our approach that does not impose this requirement. Moreover, we do not have to use run-time values of input data to establish a prediction. As can be expected, however, our predictions are thus less precise. Our basic idea is to statically predict execution times to improve static task mapping, for which our predictions are sufficiently precise, and not requiring to profile current applications to obtain a prediction is hence the greater benefit.

Guim et al. [GRCG08] try to improve job scheduling policies on large distributed architectures, called *grids*, via machine learned execution time predictions based on static information provided by the user. When a job with a static description is submitted to the scheduler, it computes all possible allocations to current idle execution nodes of the grid architecture where the job would be able to execute. For each allocation the expected execution time is estimated with a prediction model based on *classification trees*. Features used by the algorithm to build the tree merely are the executable name, number of requested processors, user ID, group ID, and the site where the job is submitted. Hence, predictions depend mainly on user behavior instead of

properties of executed instructions. Since they have used classification trees, a prediction is an interval of possible execution times. The authors have iteratively determined the range of each interval by configuring their ML tool WEKA (Waikato Environment for Knowledge Analysis), an open source Java package, to automatically discretize the execution time according to the performance obtained in the evaluation of resulting trees. This yields only five classes with large ranges: the first class contains execution times up to about 12500 units (the authors do not mention the scale), each of classes two to four comprises about 12000 units, and class five contains all execution times greater than about 49500 units. In contrast, execution times are not discretized by our approach, thus yielding more fine grained predictions. To evaluate their approach, log files with execution traces collected from the target architecture during five months are used to build the model. No other information is given about used workloads and actually observed execution times. Their experimental evaluation using cross validation shows a large mean prediction error of 160%.

The proposed method of Gupta et al. [GMD08] also predicts execution times in the form of time ranges with a prediction model based on *classification trees*, which they called *Predicting Query Run-time* (*PQR*) trees. Predictions are made for queries on enterprise data warehouses to improve its workload management. First, a query optimizer outputs an execution plan for a query and an estimate of the query cost. Both information together with a load feature vector extracted by a load monitor of the database are input to the PQR prediction model. The authors do not exactly term which features from the query execution plan and the load vector are used by their learning algorithm. They compute some derived attributes as features of the query plan and multiply them by the number of current queries running on the grid at submission of the query to obtain some features for the load vector. When a new query is submitted to the system, PQR hence estimates the execution time of the query under current load conditions. This estimate is passed on to the workload manager, which schedules the queries. Different from original classification trees where the classes (or time ranges) are known prior to the creation of the tree, at every node of the PQR tree a classifier is determined that predicts the two subranges of the current range for the children. As the PQR tree grows, the classes are automatically obtained by a combination of a classifier and two subranges at every node that gives the highest accuracy, i. e., the greatest fraction of correctly classified queries. As possible classifiers for the nodes of the PQR tree, the authors have used k-NN classifiers with $k = 1$ and $k = 3$, and *classification trees* with six different model parameters. They use different configurations for building the tree, such as possible candidates and the number of the largest time gaps as the points at which to partition the time range into two or the minimum accuracy and minimal interval size as threshold for further partitioning the current range. For their experiments, they have used up to 1000 queries, 90% of these to build the tree and 10% to predict their execution time. Different configurations yield trees with 1 to 16 classes and a mean accuracy of at least 80%. For our approach, we have decided not to classify the execution time to obtain more fine grained predictions. It

is, moreover, unclear how their approach for queries in databases with features not specified in greater detail can be transferred to computational tasks.

Matsunaga and Fortes [MF10] have extended the PQR tree approach of Gupta et al. [GMD08] to the regression problem by allowing the leaves of the tree to select the best regression method. For building the tree, called *PQR2* tree, nodes are constructed using the method of Gupta et al. [GMD08]. That is, a combination of a classifier and two subranges with highest accuracy at every node determines the ranges for its children. At leaves of the tree, PQR2 automatically selects the best regression model from a pool of basic regression algorithms. The pool used in their experiments comprises *linear regression* and *Support Vector Machine* (*SVM*) models, but other models can be placed in the pool as well. The authors have evaluated two bioinformatics applications, namely Basic Local Alignment Search Tool (BLAST) and Randomized Axelerated Maximum Likelihood (RAxML), to demonstrate the accuracy of PQR2 against other learning algorithms like decision trees, neural networks, SVMs, and k-NNs. For each application, they have used different features holding properties about the target architecture and arguments given to the application at run-time as input to their learning algorithm. In the BLAST scenario, considered features are cluster name, CPU clock, amount of memory, location of data (a nominal value indicating the name of the resource hosting the data), and CPU, memory, and disk speed, together with the number of bases in a sequence as run-time argument. In the RAxML case, cluster name, number of threads, CPU clock, amount of cache and memory, and CPU, memory, and disk speed, together with taxa size (a taxon is a group of populations of organisms), number of bases in a sequence, and taxa size multiplied by number of bases as input size are considered. Hence, their approach is application dependent and must be configured by an application expert to allow for prediction with applications different to BLAST and RAxML, which is a major drawback of their method compared to our approach. All PQR2 models created during their experiments have a clear tendency to select SVM as the leaf regressor. The overall evaluation shows a lower mean percentage error when compared to other learning algorithms. We have adapted the PQR2 approach of Matsunaga and Fortes [MF10] to our work. We exclusively use static code features of applications as input to our learning algorithm and we additionally have employed *Naïve Bayes* as possible classifier at inner nodes. Furthermore, instead of using only k-NN classifiers with $k = 1$ and $k = 3$ at inner nodes, we use a ten range grid search for the best k between $k = 1$ and $k = 10$. We have then evaluated if the PQR2 tree also yields the lowest error compared to other learning algorithms to decide whether it is advantage to employ this technique for our static setting (see Section 7.2.3).

The approach of Huang et al. [HJY+10] aims at predicting execution times for computer programs using polynomial regression. It automatically extracts a few from hundreds of retrieved features from program execution to construct an explicitly sparse model. Their method consists of four steps. First, a feature instrumentation step analyzes the source code and automatically instruments it to extract values of program features that influence the execution time, such as loop iteration counts, branch counts (how many times each branch of a

conditional jump has executed), and variable values. Second, a profiling step executes the instrumented program with sample input data to collect values for all created program features and execution times. Third, a slicing step analyzes each automatically identified feature to determine the smallest subset of the current program that can compute the value of that feature, the so called *feature slice*. This is the cost of obtaining the value of the feature – if the whole program must execute to compute the value, then the feature is expensive and not useful, since profiling can just measure execution time and there is no need for prediction, whereas if only a little of the program must execute, the feature is cheap and therefore possibly valuable in a predictive model. The final step uses the feature values collected during profiling along with the feature costs computed during slicing to build a predictive model on a small subset of generated features. To obtain a model consisting of low-cost features, their method iterates over the modeling and slicing steps, evaluating the cost of selected features and rejecting expensive ones, until only low-cost features are selected to construct the prediction model. At run-time, given a new input, the selected features are computed using the corresponding slices, and the model is used to predict execution time from the feature values. They evaluate the accuracy of their approach with three programs, the Lucene text search software and two image processing programs for finding maxima and for segmenting an image. Lucene has been executed 3840 times with different text input queries and each image processing algorithm has been executed 3045 times with diverse images. Their method automatically generates 126 features for Lucene for each execution, 182 features for the former image processing algorithm, and a large number of 816 features for the latter. They randomly split every data set into a training set and a test set for a given training-set fraction, train the algorithms, measure their prediction error on the test data, and report mean percentage errors for training-set fractions between 0.05 and 0.5. On average, their method achieves errors between 7% and 13% at fraction 0.05 and errors between 5% and 8% at fraction 0.5. However, they do not mention how many features of the large feature space were automatically selected during this experiment. Using application dependent features that hold properties of run-time behavior during execution has the drawback that for each new application, a data set of at least 300 to 400 profiling runs must be collected to build a precise model. Additionally, to predict execution times there is still the need to profile, albeit only slices of the program must be executed. The authors do not mention the average amount of executed instructions for slices versus those for whole program execution, or their needed execution time versus full profiling time. It is hence unclear how beneficial their approach is compared to profiling.

Recently, Priya et al. [PdSRdC13] present an approach for predicting execution times of ML tasks, namely classification algorithms, with the use of regression modeling. They study the accuracy of four regression models, *linear regression*, *decision trees*, *k-NN*, and *SVM*. Input to these regressors are features characterizing the data sets given to classification algorithms and the current state of the target architecture. As data set characteristics they use the number of instances in the data set, number of classes, number of attributes (features for the classification algorithm) and proportion of continues

attributes, class entropy, and the absolute correlation between all pairs of attributes and between execution time and attributes. As measure for the current state of the architecture they use current free memory available and whether the current CPU is idle. During experiments they have evaluated the accuracy of regressors for predicting execution times for six classification algorithms over 78 data sets using the mean absolute prediction deviation from the actual execution time as precision measure. As baseline method to compare with, they use the median absolute execution time in the training set. The majority of considered regressors are able to generate more accurate predictions than this baseline method. The predictions deviate from the actual time between 0.48 and 8.1 seconds on average. Their approach shows the ability of regression modeling to predict execution times. However, it is tailored to ML tasks and hence cannot be applied as general as our approach. Furthermore, since it depends on input data sets, for a new type of computer or ML algorithm, new experiments need to be run for the data sets in order to estimate new running times.

We have discussed in this section related work for predicting execution times based on previous observations. All discussed approaches require certain run-time information to be known since they use it as input to their prediction techniques. In contrast, our approach does not impose this requirement because our machine learned predictor is statically applicable. In Section 3.2, we have discussed related work for improving WCET analysis for hard real time systems that demand absolute guarantees on program execution time. These techniques also do not require knowledge of run-time information. In this section, however, we do not discuss in detail further work in this area because we propose a compiler framework for mapping arbitrary real-world concurrent applications to parallel architectures for which WCET analysis is inappropriate (see, e. g., the work of Lisper et al. [LES+13] that reports disadvantages and workarounds of problems when attempting to analyze the WCET). There exist recent approaches for WCET analysis based on AI, such as [LYGY10, GGL12, MU12, RÅdSF13, CR13]. However, AI might slow down the analysis such that it becomes impractical. In particular, this can be observed for the analysis of program loops with high iteration counts for which each loop iteration is separately simulated [LCFM09]. Recently, a WCET analysis based on AI and *Integer Linear Programming* (*ILP*) is proposed by Chattopadhyay et al. [CKR+12]. However, additionally to the drawbacks of AI, ILP may suffer from the explosion in the number of generated ILP constraints. Their analysis takes up to about 300 seconds, with an average of 20 to 30 seconds over all programs, which is unacceptable for compilation of real-world concurrent applications.

3.5 Task Mapping

With this thesis, we aim at improving task mapping of concurrent applications to parallel architectures. In past decades, plenty of approaches for task mapping with different goals and techniques were proposed. A recent survey

of Sahu and Chattopadhyay [SC13] presents application mapping strategies in detail, classifies mapping algorithms into different categories and compares them. In general, mapping algorithms aim at improving performance of concurrent applications, i. e., reducing their completion time. Beyond this objective, mapping techniques may minimize other objective functions at the same time. Many recent techniques such as [OKS07, GSH09, HBRK11, AAJ+11, MMK+11, GZM+12, ZG13, TKD12, KKW+12, AASA13, RV13] additionally aim at reducing power consumption, i. e., they propose *Dynamic Voltage and Frequency Scaling* techniques (DVFS) for an energy efficient mapping. Several approaches like [RGB+06, DLR+07, LYH10, GDTF+12, NC12, HNY12, DKV13a, AASA13, CLA13, WLQS13] are communication-aware. That is, their additional objective is to allocate parallelly executable tasks so that overall communication cost is minimized. Some approaches are thermal-aware, for example [CRW07, CHD11, TSYB11, TJS+13]. They migrate tasks from hot processors to cooler ones or perform DVFS for thermal management to mitigate problems induced by thermal hot spots, such as high cooling costs or microprocessor failure rate due to electron migration. Several proposed techniques like [FDM+08, WB11, NC12, GDTF+12, JMBT12, CLA13] are memory-aware. Their additional objective is the optimization of memory accesses in the presence of distributed memory with constrained sizes. Certain recent approaches such as [HTM10, HYX11, CM11, DKV13a, DKV13b] are reliability-aware. They propose system-level fault-tolerant techniques to extend the lifetime of multiprocessor systems or to improve the lifetime in terms of mean time to failure.

To attain their objective functions, approaches employ different methods. Many mapping strategies propose to use ILP (e. g., [RGB+06, OKS07, YH08, GSH09, HBRK11, NC12, JKEM12, WLQS13]) or employ genetic/evolutionary algorithms (e. g., [PLL07, GTT09, Yoo09, YH09, MMK+11, KKW+12, HNB+12, YYWL13, AASA13]). However, a major drawback of ILP is its limitation to small problems due to its computational complexity, and the main disadvantage of genetic or evolutionary algorithms is the extended compile time due to the iterative approach of finding good solutions. The majority of proposed techniques takes as input a specification of the application as annotated graph (e. g., [RGB+06, DLR+07, PLL07, FDM+08, YH08, GTT09, Yoo09, YH09, LYH10, HTM10, HYX11, CHD11, TSYB11, AAJ+11, MMK+11, WB11, KKW+12, TKD12, GZM+12, NC12, HNB+12, JKEM12, JMBT12, DKV13a, DKV13b, AASA13, RV13, WLQS13]). That is, they assume that applications are already transformed into a graph representation with annotations of necessary values about run-time behavior. Common to all these techniques is the requirement of having information about communication overhead or execution time of the tasks. Thus, our technique supports these approaches by automatically generating predictions for requested information. Several approaches are tailored to special target hardware like GPUs (see survey [OLG+07] that summarizes research in mapping general-purpose computation to graphics hardware), MPSoCs (e. g., [RGB+06, CRW07, OKS07, GSH09, LYH10, CHD11, HBRK11, AAJ+11, WB11, ZG13, TKD12, DKV13a, DKV13b, AASA13, RV13, CLA13, WLQS13]), or reconfigurable architectures (e. g., [HNB+12, HNY12, TJS+13, YYWL13]). Each one

has its justification, but no work has been done that is as general as our approach. For arbitrary concurrent applications starting from source code without annotations, we automatically estimate necessary run-time behavior and map parallelly executable tasks to any target architecture. Hence, we see our approach not as a competitor with the presented ones, but as an extension. For example, all communication-aware approaches that take an annotated graph representation as input to their mapping scheme can be extended with our framework by automatically generating a graph representation and providing predictions for required annotations from the source code. Consequently, this related work can be applied on real-world programs of different application domains by employing our framework.

3.6 Summary

In this chapter, we have given an overview of related work regarding the use of ML techniques in compilers, the prediction of run-time behavior, and the challenge of task mapping. For each area in turn, we have discussed why previous work is not sufficient for our purposes. Using ML techniques in compiler construction aims at improving certain compiler optimizations or optimization passes. In contrast, we apply ML techniques to statically predict loop iteration counts, recursion frequencies, and execution times. We have discussed drawbacks of existing approaches that also deliver predictions for this run-time behavior, and we have highlighted the advantages by applying our framework instead. Our proposed ML techniques automatically generate predictors, which relate static code features to the dynamic behavior, thus providing the compiler with intelligence. At compile time, these predictors can be used as precise and fast heuristics for communication-aware task mapping and power minimization, which eliminates the need for manual annotations of forecast values or profiling at compile time. Regarding task mapping, no work has been done that is as generally applicable and fully automatic as our proposed framework. For example, most of previous work assume an annotated graph representation of the application to be mapped and are tailored to special target architectures. Instead, our general framework can be instantiated for a wide diversity of parallel programming models and parallel architectures, and it does not require any annotations by the user.

4 A General Framework for Machine Learning based Mapping

A vital part of mapping concurrent applications to parallel architectures is the allocation of parallelly executable tasks of the application to *Processing Elements* (*PEs*) of the target architecture. This requires predictions for the execution times of tasks and for the communication between tasks, whose overheads have emerged as the major performance limitation. For the latter, estimates for loop iteration counts and recursion frequencies are needed if communication arises within loops or recursive functions, respectively. These predictions have to be derived automatically to facilitate an automatic mapping. Furthermore, improving an initial mapping or improving static mapping techniques requires to predict the run-time behavior in advance. Since purely static analyses must consider all cases of possible run-time behavior, they may drastically over-approximate considered behavior and hence their predictions are imprecise. To overcome this, profiling and manual annotations of forecast values were proposed – both having certain drawbacks. Regarding profiling, its result must be fed back to an additionally needed compilation step after executions of the application that deliver the run-time behavior, called *feedback directed optimization* (*FDO*). This involves instrumentation of code during an initial compilation step to specify run-time information that has to be collected. Then, instrumented code must be executed sufficiently many times with typical input data for each execution until all information is collected. Since profiling additionally executes operations for analyzing and storing necessary run-time values to derive needed information, it may lead to a substantial execution time overhead. Regarding manual annotations, it is difficult and error-prone or often an impossible burden for the programmer since not all run-time information can be known.

In this chapter, we present our framework for *Machine Learning based* ***Ma****pping of* ***C****oncurrent* ***A****pplications to* ***P****arallel* ***A****rchitectures* (*MaCAPA*). MaCAPA[1] eliminates the need for manual annotations of forecast values or profiling at compile time by automatically generating *Machine Learning* (*ML*) based predictors for requested information about the run-time behavior of par-

[1] Macapá is also the capital of Amapá state in Brazil.

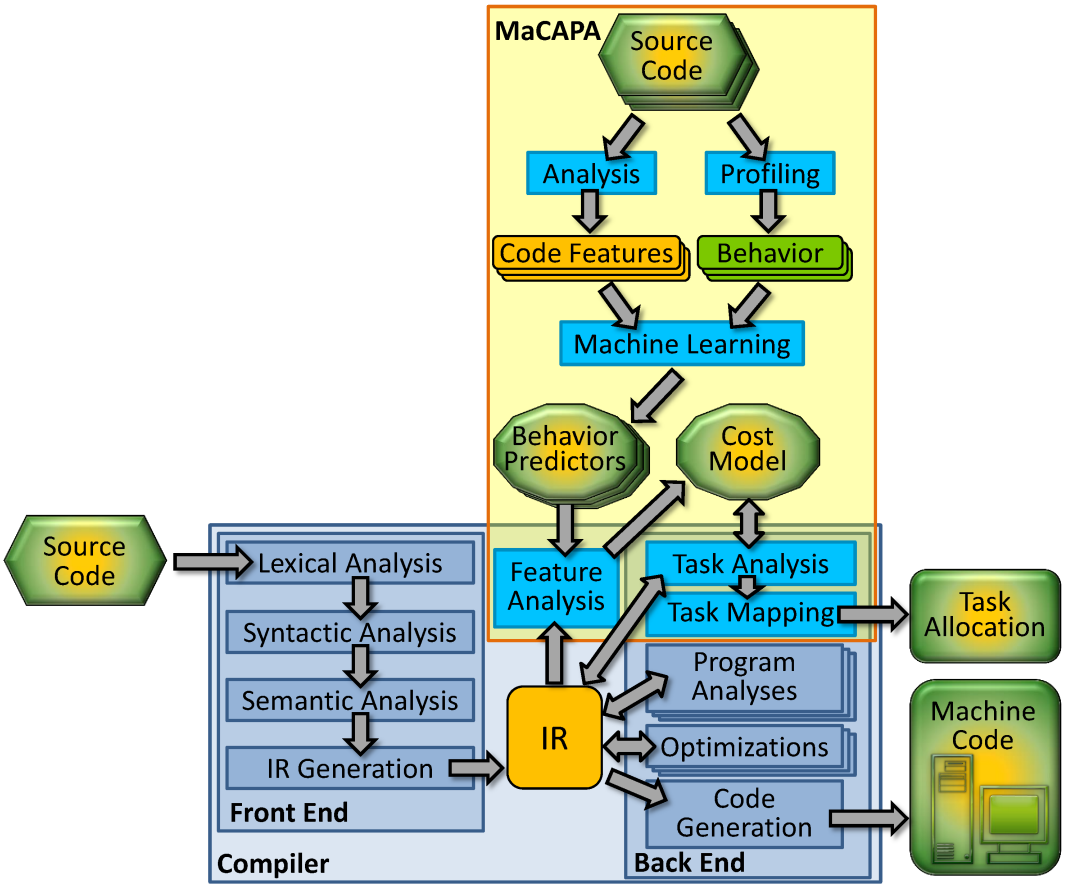


Figure 4.1: Compiler extended with our *MaCAPA* framework

allelly executable tasks. Each predictor solely uses static code features of the compiled program to derive its prediction. This bridges the gap between static program analyses on the one hand and dynamic run-time behavior on the other hand since our beforehand machine learned predictors relate static code features to observations from several profiling runs.

4.1 Overview

We propose a general framework MaCAPA that automatically provides a compiler with knowledge of run-time behavior of programs and, based on this, maps parallelly executable tasks of applications to PEs of the target architecture. Thereby, our framework MaCAPA minimizes predicted communication overhead between tasks and overall execution time of the application. The basic idea of our framework is to derive predictions about unknown run-time behavior via ML techniques that relate static code features to dynamic run-time behavior, thus making the compiler *intelligent*. Existing compilers can easily be extended with our framework. Figure 4.1 illustrates our approach. At the bottom, we see a conventional compiler. It first converts source code that is consistent with the syntax and semantics of a programming language to an *Intermediate Representation* (*IR*). Then, optimizations on this IR take place,

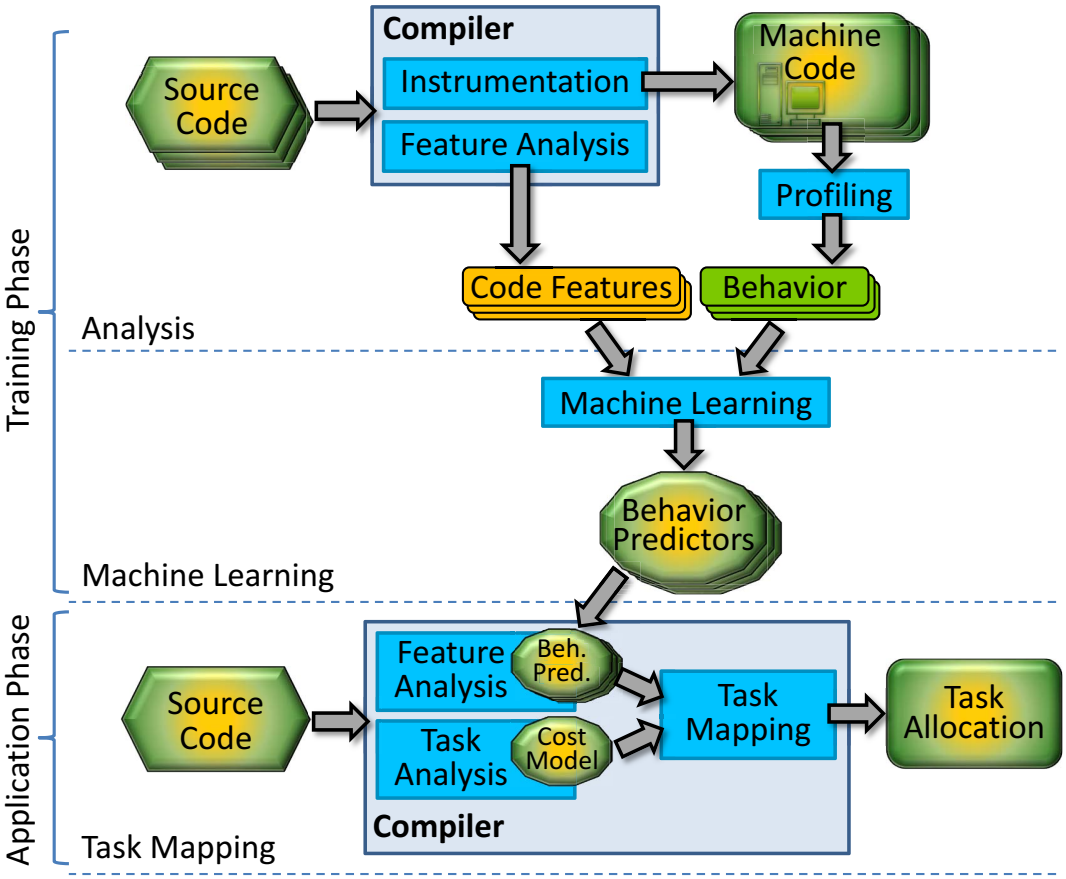


Figure 4.2: Phases in *MaCAPA*

aiming at improving the representation w. r. t. an objective function. These optimizations use results of program analyses to ensure program correctness and to decide which transformation will be applied. Our framework MaCAPA extends conventional compilers by supervised ML techniques, automatic analyses and a task mapping technique. Our ML techniques yield statistical models that approximate the relationship between static code features and regarded run-time behavior as closely as possible. Because the ML techniques require a set of observations as training data to determine the relationship, MaCAPA automatically analyzes necessary code features and profiles run-time behavior from a comprehensive suite of programs. As a complement to purely static program analyses, our framework MaCAPA automatically provides the compiler with these machine learned models as predictors for unknown run-time behavior. Since the behavior predictors need static code features to derive predictions for the current program, our framework extends the compiler by corresponding feature analyses. A task analysis determines executed instructions of tasks and interdependencies between tasks. Based on this, our framework automatically derives a precise cost model to perform communication- and power-aware task mapping.

Our framework MaCAPA comprises two basic phases (see Figure 4.2). To obtain behavior predictors based on ML, a one-off *Training Phase* per architecture prior to the actual compilation of programs is required, which comprises an

Analysis Phase and a *Machine Learning Phase*. The compilation of programs takes place during the *Application Phase* of MaCAPA. Since the *Training Phase* is decoupled from actual compilation, the compile time is not extended by training. During the *Analysis Phase*, we collect static code properties and dynamic run-time behavior via profiling from a comprehensive suite of programs to obtain training data. For this purpose, we provide a feature analysis and a module that instruments the code to analyze and store run-time information. Code features and run-time behavior both are input to the *Machine Learning Phase* where ML models are automatically constructed that relate static code features to observed run-time behavior. These models are used as predictors for unknown run-time behavior to assess the cost of alternative task mappings during the *Application Phase* of our framework. During compilation, it is only necessary to statically analyze a number of code features to obtain a prediction. These feature analyses are typically less computationally intensive than corresponding program analyses. Hence, our approach has a negligible compile time overhead. Our automatic task analysis determines instructions to be executed by each task as well as the communication and interdependencies between tasks. On this basis, a cost model is constructed that employs obtained behavior predictions to derive computational costs of tasks and communication costs between tasks. Our task mapping technique then uses the cost model to rate the gain of alternative task mappings. As a result, the allocation of parallelly executable tasks to PEs of the target architecture can be improved since it is based on predictions of expectable run-time behavior.

In the following sections, we describe each phase of MaCAPA in more detail. In Section 4.2, we start with the *Analysis Phase* that comprises feature analysis and profiling run-time behavior. In Section 4.3, we present the *Machine Learning Phase* that yields behavior predictors. In Section 4.4, we introduce our static task analysis and we show how obtained predictors and our cost model are used to improve task mapping during the *Application Phase* of MaCAPA.

4.2 Analysis of Applications

In the *Analysis Phase* of MaCAPA, we collect static code features from a comprehensive suite of programs. Additionally, observations of run-time behavior have to be collected since we use supervised learning techniques. Both types of information constitute the required training data as input for our ML algorithms during the *Machine Learning Phase* of MaCAPA. We aim at learning predictors for different run-time behavior, thereby using different ML techniques. We therefore analyze several code features and compose different feature vectors for each regarded run-time behavior. To obtain run-time information, we automatically instrument the code of the program to analyze and store values of regarded behavior and we execute instrumented code with representative input data of the program. We describe both steps of this *Analysis Phase* in the following. Our feature analysis is given in Subsection 4.2.1 and our profiling approach is given in Subsection 4.2.2.

4.2.1 Analysis of Static Code Features

In this subsection, we present the static code features used for learning the regarded run-time behavior. In particular, our learning algorithms aim at achieving four goals, namely learning loop iteration counts, recursion frequencies of functions, execution times of tasks, and the PE in a given architectural setting that executes a task most efficiently. In the following, we first describe in detail the feature vectors for learning loop iteration counts, recursion frequencies of functions, and execution times of tasks. Afterwards, we discuss the features for learning the PE that executes a task most efficiently.

Features for Loop Iteration Counts

To learn statically undeterminable loop iteration counts, we consider static code features, which hold characteristics of the loop structure, the loop bounds, the loop body, and the function that contains the loop. In total, we use 114 code features (see Table 4.1 for a complete list of features). In the following, we describe the rationale for this choice of features.

Loop structure (features 1 – 5) We consider code features such as the kind of the loop and the loop exit branches. We distinguish as kind of loops between `while-do` or `for` loops and `do-while` loops because the latter ones iterate at least once. We examine the kind of exit branches to differentiate between exits that stem from the loop bound, exits that stem from `break` statements, and exits that arise from `goto` statements. We also consider the number of exit branches out of the loop. The more exit branches a loop contains (e.g., due to `continue` or `break` statements), the greater the likelihood that the loop iteration is actually terminated because each exit has a certain branch probability, which sum up to this likelihood. We also take the nesting of loops into account because iterations of inner loops may depend on induction variables of outer loops.

Loop bound (features 6 – 63) We also examine characteristics of the structure of loop bounds like, for instance, the number of con- and disjunctions and (un-)equality comparisons between pointers or values. The more conjunctions a loop bound contains at the top level, the greater the likelihood that the loop will iterate less often. This statistical assumption holds because each term within the conjunctions defines a constraint on continuing the loop iteration and their conjunction thus tightens the loop bound. Parallel to this, the more disjunctions a loop bound contains at the top level, the greater the likelihood that the loop will iterate more often. Seeing that basic data types and hardware addresses span a great range of values, the probability of two values being equal is low. Hence, a comparison of values or pointers as to whether they are equal (or unequal) within a loop bound will probably result in less (or more) iterations.

Table 4.1: Static code features for learning loop iteration counts

	No.	Name	Description
Loop structure	1 – 2	kind-{struc,exit}	kind of loop structure and exit branches
	3	cnt-exit	number of exit branches
	4 – 5	nest, max-nest	nesting of current loop, max. nesting of contained inner loops
Loop bound	6 – 8	cnt-{log,or,and}	number of logical terms, dis-, and conjunctions in loop bound
	9 – 20	cnt-{i,f}-{l,le,g,ge,eq,ne}	number of comparisons of integer / floating-points to be less / less or equal / greater / greater or equal / equal / not equal
	21 – 32	cnt-{iz,fz}-{l,le,g,ge,eq,ne}	ditto when compared to zero
	33 – 44	cnt-{im,fm}-{l,le,g,ge,eq,ne}	ditto when compared to minus one
	45 – 56	cnt-{ic,fc}-{l,le,g,ge,eq,ne}	ditto when compared to another constant value
	57 – 62	cnt-ptr-{l,le,g,ge,eq,ne}	ditto when pointer are compared
	63	cnt-ptr-nil	number of comparisons of pointer against `NULL`
Loop body	64 – 69	{min,max}-sz-{arr,rec,un}	min. / max. size of referenced arrays, records, unions
	70 – 75	{min,max}-bsz-{arr,rec,un}	ditto when ref. in loop bound
	76 – 78	cnt-{arr,rec,un}	number of accesses to elements of arrays, records, unions
	79 – 86	cnt-{f}{scanf,printf,getc,putc}	number of calls to these functions
	87 – 94	w-{f}{scanf,printf,getc,putc}	ditto weighted by static execution frequency
	95 – 97	cnt-{fopen,fclose,fflush}	number of calls to these functions
	98 – 100	w-{fopen,fclose,fflush}	ditto weighted by static execution frequency
	101 – 103	cnt-{bb,ass,expr}	number of basic blocks, assignments, expressions
	104 – 106	w-cnt-{bb,ass,expr}	ditto weighted by static execution frequency
	107 – 109	frac-{call,ctrl,if}	fraction of function calls, statements altering the control flow, `if`-expressions
	110 – 112	w-frac-{call,ctrl,if}	ditto weighted by static execution frequency
Env.	113	cc	cyclomatic complexity [McC76]
	114	nd	nesting depth [GS85]

Loop body (features 64 – 112) If arrays are referenced within a loop body, the loop can be expected to iterate over a certain number of its elements. We therefore consider the maximum size of the referenced arrays as a static feature. How many times the loop iterates over the elements is ascertained by the profiling runs whose results are fed into the learning algorithm. We also take function calls within the loop body into account. For example, file-IO operations with characters may result in more loop iterations than file-IO operations with strings for the same file. We use the number of *basic blocks* (*BBs*), assignments, and expression within assignments together with the fraction of function calls and (conditional) branches in the *Control Flow Graph* (*CFG*) as measurement how complex the computations of the loop are and which fraction of the body is executed.

Loop environment (features 113 – 114) As characteristics of the function that contains the loop, we use the cyclomatic complexity [McC76] and the nesting depth [GS85]. This indicates how complex the function is compared to the complexity of the loop. We additionally consider the static prediction of execution frequencies proposed by Wu and Larus [WL94] as a third metric. Since it is a good estimation of how often a loop will execute, it is used to weight several features by multiplying them with the value of the predicted execution frequency.

In summary, the feature vector $\mathbf{x}$ of our training set

$$X = \{(\mathbf{r}^t, \mathbf{x}^t) | t = 1, \ldots, N\}$$

for learning the loop iteration count has d=114 dimensions. To evaluate our approach, we have used $N = 24880$ loops for training the predictors. Thereby, we have observed iteration counts up to over 4 billion. Because we have used the truncated decadic logarithm as classification (see Subsection 7.2.1), $\mathbf{r}$ has 10 dimensions.

Features for Recursion Frequencies

To predict recursion frequencies, we use 19 code features in total (see Table 4.2 for a complete list of features). The features hold the structure of the recursive function and characteristics concerning parameters, return values, and used variables, their size and arithmetic operations on them. The rationale is the following.

Function structure (features 1 – 3) We consider the static number of (recursive) self calls within the function body as feature because it directly influences the recursion frequency. A descriptive example is the Fibonacci function with two self calls compared to the factorial function with one self call, which results in exponentially more recursions for the same input. Because one cannot expect that all self calls will be executed at each step, we weight this feature

Table 4.2: Static code features for learning recursion frequencies

	No.	Name	Description
Structure	1 – 2	c-self{a,w}	static number of self calls absolute/weighted
	3	cc	cyclomatic complexity
IO	4 – 6	rec{p,r,v}	recursive structure of parameters/return value/global variables
	7	c-charptr	number of char* as parameter
	8 – 9	file{str,char}	file-IO of strings/characters
Size	10	par-loc-sz	sum of sizes of parameters and local variables
	11 – 12	{mi,mx}arsz	min./max. size of arrays
	13 – 15	c{ar,rec,un}	number of accesses to arrays/records/unions
Arithm.	16 – 19	s{df,dv,sf,ad}	sum of values at subtractions/divisions/ shifts/address operations

with the execution frequency prediction, computed using the static branch prediction algorithm proposed by Wu and Larus [WL94]. We use the *cyclomatic complexity* by McCabe [McC76] as feature because we expect more complex functions to potentially recurse less.

IO of function (features 4 – 9) We examine parameters, return values, and used global variables whether they are a (recursive) structure containing pointers to itself. We use the static number of these pointers within each recursive structure as feature because it affects the expectable recursion frequency. For example, the recursion frequency during traversal over trees with left and right subtrees (i. e., with two static pointers) is potentially higher than during traversal over lists with only one pointer to the next element. Because recursion is heavily used for string traversal, we count parameters that are char pointers. File-IO operations with characters are distinguished from file-IO operations with strings because reading or writing a file character by character will potentially result in more recursions than doing that line by line.

Size (features 10 – 15) We also consider the size of parameters and local variables because they contribute to the size of the stack frame. The greater the stack frame, the more is the recursion frequency bound by the stack size. The minimum and maximum of statically known sizes of referenced arrays and the number of accesses to arrays, records, and unions are used as a feature because the function can be expected to iterate over all elements.

Arithmetic operations (features 16 – 19) We also examine arithmetic operations with parameters or global variables. We sum the values used at each operation if known to be constant, otherwise the value 1 is used. Assume, for instance, recursion stops at value zero of a certain variable. When the variable is shifted, the frequency is potentially less than the frequency when it is divided or when a value is subtracted from it. Likewise, if the used value is 4, it potentially results in less recursions than using the value 2.

In summary, the feature vector $\mathbf{x}$ of our training set

$$X = \{(\mathbf{r}^t, \mathbf{x}^t) | t = 1, \ldots, N\}$$

has 19 dimensions. We have used $N = 424$ functions for learning recursion frequencies. Thereby, we have observed recursion frequencies up to over 300 million and we have used the truncated decadic logarithm as classification (see Section 7.2.2). Hence, $\mathbf{r}$ has 9 dimensions.

Features for Execution Times

The execution time of a task clearly depends on executed instructions of that task. Since we aim at predicting execution times of tasks solely based on information that is available at compile time of applications, we cannot know which instructions are actually executed. The information we have is the IR obtainable from the source code, especially a CFG for each task. Lowering the IR to *Low-Level IR* (*LIR*) yields a representation where the instructions within BBs of the CFG are close to assembler code. During code generation in the back end of the compiler, machine code to be emitted from LIR is available. We consider features for learning execution times that represent these machine code instructions. Depending on the target architecture, the number of different machine code instructions that a PE can execute ranges from about ten to more than two hundred. If we chose to represent each different instruction by a different feature, we would generate a sparse feature vector in the latter case. We thus use equivalence classes of machine code instructions with respect to similar behavior to categorize instructions and we use these categories as features. The categorization is as follows (see Table 4.3).

Tests (features 1 – 2) The feature *bitwise* represents instructions on bits, e. g., tests on bits at logical comparisons like conjunctions and disjunctions or operations on bits to perform logical negation. Instructions captured by the feature *comp* are relational operators that compare arithmetic values in integer or floating point representation.

Arithmetic operations (features 3 – 6) The feature *calc* represents calculations like address arithmetic or mathematical operations on integer data types. The feature *convert* captures conversions between integer data types such as sign extensions or zero extensions. Instructions represented by the feature *fcalc*

Table 4.3: Static code features for learning execution times

	No.	Name	Description
Tests	1	bitwise	operations on bits
	2	comp	comparison between data values
Arithmetic	3	calc	integer arithmetic calculations
	4	convert	conversion between integer data types
	5	fcalc	floating point arithmetic calculations
	6	fconvert	conversion between floating point and integer values
Control	7	docall	function calls
	8	param	push actual parameters to stack
	9	jump	jumps in the control flow
Copy	10	addr	load effective address
	11	load	load of values into registers
	12	move	move values between registers
	13	store	storage of values to memory

are arithmetic operations with numbers in floating point representation. With the feature *fconvert*, we capture conversions from integer to floating point data types and vice versa.

Control flow related (features 7 – 9) Function calls, which alter the control flow, are classified by the feature *docall*. For each function call, a different number of arguments must be given to the callee. We capture instructions that push data to the stack to pass parameters of function calls with the feature *param*. The feature *jump* represents jumps in the control flow, either conditional or unconditional.

Copy instructions (features 10 – 13) The feature *addr* captures instructions that calculate the effective address of objects and store it in a register. We represent instructions that load values from memory or from its spill location into registers with the feature *load*. The feature *move* represents instructions that move values between registers or between registers and spill locations. With the feature *store*, we capture instructions that store values from registers to memory.

Given the CFG of a task, we visit all BBs and record categorized instructions within each BB in the feature vector. Since not all instructions are actually executed due to conditional jumps in the CFG, we do not simply

count categorized instructions. Instead, we weight each occurrence with its execution frequency.

We have developed two different weighting schemes. The scheme to be used when solely static information is available multiplies each occurrence with the static execution frequency prediction that we compute according to the algorithm of Wu and Larus [WL94]. That is, if the algorithm predicts that a BB will be executed, e. g., 2 times, we multiply the number of each instruction within the BB with 2 and add the result to the corresponding category.

For the other scheme, we also weight each occurrence of instructions. As weighting factor, we take the execution frequency that we derive from interprocedural path profile information (see Subsection 4.2.2 for the algorithm). This scheme is not applicable in a purely static setting since run-time information from profiling is required at compile time. We use it however for experimental evaluation whether our chosen instruction categorization is basically suitable for learning execution times. Nevertheless, a user of our framework can decide to use this scheme to obtain more precise results, though having the cost to perform path profiling before.

If a function is called by the task for which a prediction of its execution time is required, the executed instructions of this function also contribute to the execution time. We therefore take account of all function calls. When we know the instructions within the function body of a callee (because this function is defined in the application under compilation), we do not increase the feature *docall*. Instead, we add the feature vector of the function to the one of the task. Again, we weight the feature vector of the function with the execution frequency of the function call before summing the features to the ones of the task in order to regard that not all function calls are actually executed. Calls to functions where we cannot derive its executed instructions (e. g., functions defined in included libraries or recursive function invocations) are captured with the feature *docall*.

Features for Learning the Best Performing PE

We aim at learning the PE of a heterogeneous architecture that executes a task most efficiently. To that end, we propose two basic variants (the second variant is not yet present in the instantiation of our framework for improving the mapping of MPI programs to processor networks). The first variant is to establish machine learned predictors for each different PE with the same features as used for execution time learning, which we have discussed in the previous subsection. This automatically yields the PE that executes a task most efficiently by choosing the PE with the lowest execution time prediction. The second variant is to extend the feature vector for learning execution times with further features. Using this variant, only one ML model must be learned. To learn the best performing PE using the second variant, potential differences of PEs and their effects on the execution time of tasks must be taken into account. We represent the potential differences of PEs with *architectural features* and the architecture-dependent behavior of tasks with *architecture-dependent*

static code features (see Table 4.4). In the following, we discuss the rationale for the proposed choice of these features.

Architectural features (features 1 – 22) We propose to use the frequency at which a PE is running, called the *clock rate*, as most important feature to distinguish PEs in terms of potential performance for executing computational tasks. However, considering solely the clock rate of a PE does not indicate how many instructions can be executed by the PE per clock cycle since instructions may take more than one clock cycle to complete execution. In this case, subsequent instructions that need the result of the previous operation must be stalled. To complete more instructions per clock cycle, architectural designs may include multiple functional units working in parallel, such as *Arithmetic Logic Units* (*ALUs*), *Floating-Point Units* (*FPUs*), and *Load/Store Units* (*LSUs*). We therefore consider the numbers of these functional units as features.

Other techniques to increase the throughput of instructions that we use as features are *instruction pipelining*, *out-of-order* execution, and *speculative* execution. Instruction pipelining refers to breaking down the execution pathway into discrete stages that are performed in parallel. Common stages are fetching the instruction from memory, decoding the instruction and fetching needed data into registers, executing the instruction, and writing the results of the instruction to processor registers or to memory. The number of stages of a pipeline is called the pipeline depth, which we consider as a feature. Many architecture designs include pipelines up to 20 stages. A PE that performs out-of-order execution of instructions can avoid stalls that occur when the data needed to perform an operation is unavailable. The cycles for stalling that would otherwise be wasted are filled with instructions which are ready to execute. Then, the results are re-ordered at the end to make it appear as if the instructions were processed as normal. In case of a conditional jump in the CFG, either the jump is taken (if the condition evaluates to true) and instructions from the target of the jump must be executed or the jump is not taken and execution continues with instructions following immediately after the jump. Since it is not known whether the conditional jump will be taken until the condition is evaluated, the PE would have to wait until the conditional jump instruction has passed the execute stage before the next instruction can enter the fetch stage in the pipeline. A PE capable to perform speculative execution fetches and executes instructions of the branch that is predicted to be the most likely. If it is later detected that the prediction was wrong, then the speculatively executed instructions are discarded and the pipeline starts over with the correct branch. We propose to use features that indicate whether a PE performs out-of-order or speculative execution because both techniques may shorten the execution time of tasks.

Since a PE executes instructions and processes data that are stored in main memory, the next important features are the latency for accessing the main memory and the bandwidth of the memory bus. The bandwidth of the bus is determined by the product of the width of its data path, its clock frequency

Table 4.4: Static code features for learning the best performing PE

	No.	Name	Description
Architectural Features	1	clock	clock rate
	2	alu	number of *Arithmetic Logic Units* (*ALUs*)
	3	fpu	number of *Floating-Point Units* (*FPUs*)
	4	lsu	number of *Load/Store Units* (*LSUs*)
	5	pipe	pipeline depth
	6	ooe	whether the PE performs out-of-order execution
	7	spec	whether the PE performs speculative execution
	8	mem	latency to main memory
	9	bwidth	bandwidth of memory bus
	10	local	size of local memory
	11	reg	number of registers
	12 – 15	l1d,l2d,l3d,l4d	size of L1, L2, L3, and L4 data caches
	16 – 18	l1i,l2i,l3i	size of L1, L2, and L3 instruction caches
	19 – 20	tlb1d,tlb2d	size of translation lookaside buffers for data
	21 – 22	tlb1i,tlb2i	size of translation lookaside buffers for instructions
Architecture-dependent Static Code Features	23	max-loop-instr	maximal number of instructions within loop bodies
	24	avg-loop-instr	average number of instructions within loop bodies
	25	max-var	maximal number of variables accessed within loops
	26	avg-var	average number of variables accessed within loops
	27	max-struct-sz	maximal size of structures accessed within loops
	28	avg-struct-sz	average size of structures accessed within loops
	29	frc-int	fraction of integer operations
	30	frc-fp	fraction of floating-point operations
	31	frc-jmp	fraction of conditional jumps in the CFG
	32	avg-loc	average number of local variables of called functions
	33	global	number of global variables

and the number of data transfers it performs per clock cycle. A memory bus with low bandwidth will cause the PE to spend significant amounts of time waiting for data to arrive from main memory. With the ability to put large numbers of transistors on one chip, it becomes feasible to integrate memory on the same die as the processor. This *local memory*, such as a scratchpad memory, has a comparatively less access latency than off-chip memory. We therefore consider the size of local memory when available. If the PE does not comprise an architectural property, we set the corresponding feature to zero. The fastest way to access data needed for instruction execution are registers. The more registers a PE has, the more values can be accessed very fast. Because the same data is often accessed repeatedly, holding frequently used values in registers shortens the execution time. Thus, we suggest to use the number of registers available on a PE as a feature.

Since the performance difference between a PE and main memory has grown, increasing amounts of high-speed memory are built directly into the processor, known as cache. Most actively used data in the main memory is just duplicated in the cache memory, which is faster, but of much smaller capacity. If requested data is in the cache, this request can thus be served faster by simply reading the cache. Otherwise, the data has to be recomputed or fetched from its original storage location. In addition to *data caches*, PEs may comprise *instruction caches* where recently executed instructions can be stored to avoid fetching and decoding them multiple times. There is a trade-off between cache latency and hit rate. Larger caches have better hit rates but longer latency. To address this trade-off, PEs may use multiple levels of caches (i. e., a cache hierarchy), with small fast caches backed up by larger slower caches. To capture possible caches and cache hierarchies, we introduce corresponding features that represent sizes of data caches (level one to four) and instruction caches (level one to three). A *translation lookaside buffer* (*TLB*) is a special cache used to map virtual and physical address spaces. The TLB improves the virtual address translation speed since a requested virtual address from the TLB matches quickly the physical address. For virtual addresses that are not present in the TLB, the translation must look up the page table, which is more expensive. Similar to caches, architectures may have distinct TLBs for data and instructions and may comprise a hierarchy of TLBs. To capture this, we consider sizes of TLBs for data and instructions with two levels.

Most of the architectural features for a PE can be automatically obtained from special processor instructions (such as the instruction `CPUID` for Intel CPUs) or diagnostic programs: the clock rate, the sizes of all caches and main memory, the bandwidth of the memory bus, as well as the type, family, and model of a CPU. Other details like the sizes of TLBs, if not given by the user, can be determined from a database that holds the values for each model.

Architecture-dependent static code features (features 23 – 33) We propose to use static code features representing properties of tasks which may have an impact on the execution time for different PEs. As discussed above, PEs may have different caches for instructions and for data. When instructions

of a task are repeatedly executed (due to loops in the CFG), it is important whether all instructions within a loop body fit in the cache. If not, least recently executed instructions are replaced by most recently ones and must be fetched and decoded at each loop iteration. We therefore suggest features holding the average and the maximal number of instructions within loop bodies. Regarding data caches, the same discussion holds for accessed data within loops. Hence, we propose features that capture the maximal and average number of variables as well as the maximal and average size of data structures accessed within loops. Since PEs can differ in the number of functional units for integer and floating point arithmetic, we consider the fraction of integer and floating-point operation as features. As discussed before, registers are the fastest way to access data but PEs have a limited number of registers. We therefore propose features that indicate register pressure. Because a task may call functions during execution, a relevant feature concerning register pressure is the average number of local variables of called functions. When a function has less local variables than the number of registers of the PE, all variables can be hold in registers, which results in faster accesses. Likewise, if the number of global variables that are accessed by a task does not exceed the number of registers, they can be stored in registers. For that reason, we also consider the number of accessed global variables as a feature.

In this subsection, we have presented the static code features used for learning the regarded run-time behavior. In particular, we have introduced the features for learning iteration counts of loops, recursion frequencies of functions, and execution times of tasks. Furthermore, we have shown which features can be used to determine the PE that executes a task most efficiently. Using supervised learning techniques, it is necessary that the correct behavior is known for establishing a ML model. In the following subsection, we define the profiling steps needed to collect the correct relevant behavior.

4.2.2 Profiling Run-time Behavior

In the *Machine Learning Phase* of MaCAPA, our goal is to learn iteration counts of loops, recursion frequencies of functions, execution times of tasks, and the best performing PE. For this purpose, we employ supervised learning techniques based on a set of training examples to learn a relation between static code features and the resulting dynamic run-time behavior. Each training example is a pair consisting of a feature vector and the corresponding correct outcome (which must be identified). Hence, we have to determine the correct execution frequency of loop bodies and recursive functions for the former two goals and the correct execution time for the latter two goals. To that end, we instrument the code of the program to analyze and store observed values of regarded behavior. After execution of instrumented code with representative input data, necessary run-time observations are available for the *Machine Learning Phase* of MaCAPA. In the following, we describe both the instrumentation of code to obtain execution frequencies and the instrumentation of code to obtain execution times.

Execution Frequencies

We determine execution frequencies of loop bodies and recursive functions via automatically instrumenting the code of a program and executing the instrumented program with representative input data. The instrumentation is based on the algorithm of Ammons et al. [ABL97], which we have extended. This algorithm yields flow and context sensitive profiles[2] and thus enables interprocedural path profiling. As we have discussed in Subsection 4.2.1, we also use execution frequencies of BBs, which we derive from these profiles, to weight features for learning execution times of tasks. Basically, arrays of counters hold for each function (i. e., for each CFG) the observed execution frequency for each path through its CFG. The steps of the algorithm developed by Ammons et al. [ABL97] are:

1. Assign an integer label to every edge in an acyclic CFG such that the sum of integers (the *path sum*) is unique along each different path from the entry to exit of a function. This path encoding algorithm ensures that each distinct path generates a unique value.

2. Insert instrumentations in an acyclic CFG to track the path sum, to initialize the array that holds execution frequencies for each path in the CFG with zeros at the *entry* BB of a function, and to update the element in this array that corresponds to the current path (the element with index *path sum*) with its current execution frequency at the *exit* BB of a function.

3. Transform a CFG of a function that contains cycles (i. e., a CFG which has an unbounded number of potential paths), into an acyclic graph with a bounded number of paths. For each back edge $b = v \rightarrow w$ in the CFG, the transformation removes b from the CFG and adds two pseudo edges in its place: one from the *entry* BB to w and one from v to the *exit* BB. The former pseudo edge corresponds to reinitializing the corresponding element in the array with zero and the latter pseudo edge corresponds to incrementing the path counter along the back edge. The algorithm can handle reducible and irreducible CFGs.

4. Insert code in the *entry* BB of each CFG such that for each called function during execution, a calling context is created which holds the array of execution frequencies.

5. Insert code such that at program termination, the calling contexts are written to a log file.

Since steps one and two are only applicable on acyclic CFGs, step three is performed before on cyclic CFGs. Then, steps one and two are performed on each acyclic CFG, labeling both original edges and pseudo edges. Step five inserts the code for writing all calling contexts to a log file into all BBs where

[2] A flow sensitive profile associates a performance metric with an acyclic path through a function and a context sensitive profile associates a metric with a path through the *Call Graph* (*CG*).

a program may terminate, e. g., before `exit` calls or at the end of the `main` function. The original algorithm by Ammons et al. [ABL97] as described above associates a frequency with each path through the *Call Graph* (*CG*). Since we use these profiles among other things for learning execution times of tasks, we have to associate separate frequencies with paths *for each individual task*. To that end, we have adapted this algorithm. At step two, we do not create an array (for each called function) because we would not be able to differentiate which task has executed the paths. Instead, we employ a structure that holds the path frequencies for each task separately. We automatically insert code that creates, initializes, and updates the structure accordingly. At step four, we use calling contexts that comprise our structure. Hence, we automatically insert code in the *entry* BB of each CFG such that our contexts are created for each called function. At step five, we automatically insert code that writes the calling contexts for each task to a separate log file. Since we aim at improving the run-time performance of any concurrent application with this thesis, tasks may be realized by threads or by processes, depending on the task-level parallel programming model. Our adapted algorithm can handle both cases, i. e., it cannot only cope with processes but also with threads. This enables us to determine the execution frequency of BBs for any task individually.

Execution Times

For our supervised ML techniques during the *Machine Learning Phase* of MaCAPA, we have to determine the correct execution times of tasks. A major challenge arises from the criteria that are implied by our objectives of MaCAPA. Since we present a fully automatic framework, the run-time information must be automatically determined. Our objective is to improve the mapping for arbitrary applications. Therefore, we have to consider the various parallel programming models that may be realized by applications. Furthermore, we do not constrain the programming languages. Hence, we have to cope with any possible implementation of algorithms, e. g., recursive solutions. We have developed an automatic instrumentation algorithm to analyze execution times of tasks that fulfills these criteria. Basically, the current time can be stored before the task starts and after the task has finished. Then, the difference between both values yields the execution time for this task.

Depending on the task-level parallel programming model (see Section 2.4), parallelly executable tasks can have different appearances. For example, tasks of programs that execute in *Single Program Multiple Data* (*SPMD*) mode are the whole program (i. e., the `main` function, whereby *environmental variables* define which part of the whole program is executed by which task). Examples are programs written in *Unified Parallel C* (*UPC*), which realizes the *Distributed Shared Memory Model* (see Figure 4.3a), or *Message Passing Interface* (*MPI*), which realizes the *Message Passing Model* (see Figure 4.3b). Hence, we cannot instrument time measurements before and after a task, we have to do it within the task at start and at the end. Other examples where a task is defined by a function are programs written in threaded C (see Fig-

```
int main(int argc, char *argv)
{
  /* MYTHREAD is thread id */
  if(MYTHREAD==0){
  /* code for thread zero */
  }else{
  /* code for other threads */
  }
}
```

(a) UPC code fragment

```
int main(int argc, char *argv)
{
  int id;
  /* starts MPI: */
  MPI_Init(&argc, &argv);
  /* get current process id: */
  MPI_Comm_rank(..., &id);
  if(id==0){
  /* code for process zero */
  }else{
  /* code for other processes */
  }
}
```

(b) MPI code fragment

```
int fst=1, snd=2;
int main(int argc, char *argv)
{
  pthread_t t1, t2;
  pthread_create(&t1, ...,
      (void*)&work, (void*)&fst);
  pthread_create(&t2, ...,
      (void*)&work, (void*)&snd);
 }

void work(void *i)
{
  int *id=(int*)i;
  if(*id==0){
  /* code for thread zero */
  }else{
  /* code for other threads */
  }
}
```

(c) Threaded C code fragment

```
int main(int argc, char *argv)
{
  #pragma omp parallel
  {
    #pragma omp single
    for(int i = 0; i < 100; i++){
      #pragma omp task
      work(...);
    }
  }
}

void work(...)
{
  /* code for tasks */
}
```

(d) OpenMP code fragment

Figure 4.3: Task creation in different programming languages

ure 4.3c) and *Open Multi-Processing* (*OpenMP*) (see Figure 4.3d)[3] that realize the *Shared Memory Model*. Note that the source code itself after the `#pragma` preprocessor statement for tasks in OpenMP does not need to be a function call but the OpenMP preprocessor always creates a wrapper function that encapsulates the code. We therefore decided to start time measurements at the

[3]OpenMP is a method of parallelization, whereby the section of code that is meant to run in parallel is marked accordingly. The preprocessor directive `#pragma omp parallel` is used to fork additional threads to carry out the work enclosed in the construct. When a thread encounters the `#pragma omp task` construct, a task is generated from the code for the associated block. The directive `#pragma omp single` specifies a code block that is executed by only one thread. Hence, the tasks in Figure 4.3d are created by one thread.

beginning and to stop them at the end of functions that represent tasks. When the CFG of this function is not given, e. g., when it is a function that is defined within a library, we also create a wrapper that calls this function.

Since we aim at improving the run-time performance of arbitrary applications, we do not prohibit that tasks are realized by recursive functions. Therefore, we cannot simply start the time measurement at each function entry. Assume a first call to function `f` results in execution time t_{f1}. When this function calls itself (a second call with e. g., execution time t_{f2}), the overall execution time is not the sum of t_{f1} and t_{f2} since the latter is included in t_{f1}. Thus, we do not start time measurements for functions where measurements are running. To achieve time measurements for arbitrary applications, a global structure **t** is inserted in the IR for each function that represents a task in the program under compilation. The structure **t** comprises the following integer variables:

time for the overall execution time of the task,

start that holds the time at function entry,

running that holds whether time measurement is active, and

calls for the number of calls to this function.

Each variable within the structure **t** is initialized with the value zero. Note that the variable **calls** to count the number of invocations is optional. It can be used to determine the average execution time per call as it is done by other profilers as well. Additionally, a local integer variable **rec** that holds whether a recursive call happens is inserted in the IR for each function that represents a task. When tasks are realized by threads that share the same address space and the same function representing the task is used by more than one thread (which is the case, e. g., for the code in Figure 4.3c), the structure **t** can be indexed with the unique ID of the thread. This enables us to measure the execution times of different threads while executing the same function.

At the beginning of the task, the number of calls to the corresponding function (the element **calls** of the global structure **t**) is incremented and the value of **running** is assigned to the local variable **rec**. Since **running** is initialized with zero, **rec** is also zero if this function call is not a recursive invocation.

Algorithm 4.1: Start and stop of time measurements for tasks

```
1 start of measurement
2 |   t.calls += 1 ;
3 |   rec = t.running ;
4 |   gettime (rec, 0, &t.running, &t.start, &t.time);
5 end

6 stop of measurement
7 |   gettime (rec, 1, &t.running, &t.start, &t.time);
8 end
```

During the time measurement, **running** is set to one. A recursive call that can only happen when we measure the time of the function is thus indicated by a value of one. Then, the time measurement starts (see Algorithm 4.1, Lines 2 to 4). The time measurement is realized by a function `gettime`, which we also introduce in the IR. Its first argument is the local variable **rec**. The zero as second argument to `gettime` indicates that the measurement should start. The other arguments are references to elements of the global structure **t** of this task. The pseudo code for `gettime` is shown in Function 4.2. If it is not a recursive invocation, the system call to `clock_gettime` is performed and the current time in nanoseconds is computed (Lines 4 and 5)[4]. If the measurement should start (the second argument **stop** is zero), the current time is assigned to **start** and **running** is set to one (Lines 7 and 8). Otherwise, the difference between the current time and the stored time at start that yields the current execution time is added to the execution time of the function and **running** is set to zero (Lines 11 and 12). The instrumented statements in Lines 2 and 13 are only part of an extended algorithm, which we describe below. We use our extended algorithm to establish a greater set of observations for learning execution times of tasks. At the end of the task, the time measurement is stopped by calling `gettime` with one as second argument (see Algorithm 4.1, Line 7). The first argument is the local variable **rec**, which holds whether the function call was a recursive invocation. In this case (i. e., when **rec** is one), the measurement must not be stopped because it was not started with this call. At locations of the program where it may terminate, e. g., before `exit` calls or at the end of the `main` function, our automatic instrumentation algorithm inserts code to stop all running time measurements and to store all values together with the corresponding function names in a file.

With the ideas described so far, we achieve an automatic instrumentation algorithm to analyze execution times of tasks that we have integrated in our compiler framework. This enables a continuous compilation flow since a user does not have to interrupt the compilation in order to use external tools. In addition to the presented time measurements for tasks, we have developed an automatic instrumentation algorithm to analyze execution times for all functions, not only for functions that represent tasks. As it is done by other profilers, it computes (additionally to the execution time for each function) the time spent for own computations within the function body and the time spent by called functions (i. e., by its callees). Thus, we have integrated a complete profiler for execution times into the compiler. We use this extended algorithm to determine execution times of all functions in several programs. This establishes a greater set of observations for learning execution times of tasks, which is crucial for the quality of the resulting ML model. Using this profiling algorithm also enables to analyze *hot spots* of the program. Basically, the extended algorithm instruments all functions with time measurements. This instrumentation generates overhead in terms of executed instructions. Hence, the execution time of functions is lengthened, where the amount of overhead depends on the number of time measurements. To subtract out

[4]Although the function `clock_gettime` is a UNIX system call, it can be ported to other operating systems.

Function 4.2: gettime

```
   Parameters : int rec, int stop, int *running, long long *start,
                long long *time
1  if (rec==0) then
2  |  total_calls++;                                  /* only in extension */
3  |  struct timespec time;
4  |  clock_gettime (CLOCK_MONOTONIC, &time);
5  |  long long nanosec = time.tv_sec*1e9 + time.tv_nsec;
6  |  if (stop==0) then
7  |  |  *start = nanosec ;
8  |  |  *running = 1 ;
9  |  else
10 |  |  long long curr_time = nanosec - *start;
11 |  |  *time += curr_time;
12 |  |  *running = 0 ;
13 |  |  last_time = curr_time;                        /* only in extension */
14 |  end
15 end
```

overhead, we quantify the overhead of one measurement and we record for each function how many measurements were performed during its execution. To that end, we insert for the following variables in the IR (additionally to the required variables mentioned above)[5]:

extern_time A global integer variable for each function that holds the time spent in other functions.

gettime_calls A global integer variable for each function counting calls to `gettime` made during execution of the function.

current_calls A local integer variable for each function that holds the number of current calls to `gettime`.

last_time One single global integer variable that holds the last measured time.

total_calls One single global integer variable counting overall calls to `gettime` during execution of the program.

The Algorithm 4.1 is extended as follows (see Algorithm 4.3). At the beginning of each function, the number of calls to `gettime` that started or stopped measurements up to this point is stored in the local variable **current_calls** (Line 4). The global variable **total_calls** is set by the function `gettime` (see Function 4.2, Line 2). Then, for all invocations of instrumented callees that are in the function body and that do not represent a recursive call, the last measured time is added to the variable **extern_time** (Line 9 of Algorithm 4.3). The global variable **last_time** is also set by the function `gettime` (see Func-

[5]These variables are also optional. If the overhead is irrelevant and a user of our framework does not want to introduce too many variables in the IR, all variables and computations to subtract out overhead can be omitted.

Algorithm 4.3: Start and stop of time measurements for functions

```
1  start of measurement
2  |  t.calls += 1 ;                                          /* original */
3  |  rec = t.running;                                        /* original */
4  |  current_calls = total_calls;
5  |  gettime (rec, 0, &t.running, &t.start, &t.time);        /* original */
6  end

7  foreach call to another instrumented function do
8  |  /* add the following statement after the call */
9  |  extern_time += last_time;
10 end

11 stop of measurement
12 |  gettime (rec, 1, &t.running, &t.start, &t.time);        /* original */
13 |  gettime_calls += total_calls − current_calls;
14 end
```

tion 4.2, Line 13). Thus, the time measured for other instrumented callees are stored in the variable **extern_time**. The execution time for callees that are not instrumented (such as library functions, which are not present in the source code) is hence pooled under the time spent for own computations. Note that the calculation of overhead and the time spent by called functions is not thread-safe due to the single global variables **last_time** and **total_calls**. However, we solely aim at establishing a greater set of observations, for which the algorithm is sufficient because we apply it to several serial benchmarks. When tasks are realized by processes, our extended algorithm yields correct results. At the end of the function, we compute how many measurements were performed during execution of the function via the difference between the overall calls to `gettime` at this point and the stored number of calls at the beginning. We add this number to the global variable **gettime_calls** (Line 13 of Algorithm 4.3) to subtract out overhead. At locations of the program where it may terminate all running time measurements are stopped. In addition to the algorithm to measure execution times of tasks, we compute the time spent for own computations within the function body via the difference between **time** and **extern_time**. Furthermore, our automatic instrumentation algorithm inserts code that computes the induced overhead of our time measurements for each function (via a recorded overhead for one measurement times the value of **gettime_calls**) and that subtracts this overhead from its execution time. Finally, all values are stored in a log file.

In this section, we have presented the *Analysis Phase* of MaCAPA. In particular, we have discussed analyses of static code features and necessary profiling steps. Both is required to obtain observations that are input for our supervised learning techniques. In the following section, we define the ML techniques that we use for establishing predictors of considered run-time behavior.

4.3 Learning of Run-time Behavior

In this section, we present the *Machine Learning Phase* of our proposed framework MaCAPA. This phase and the *Analysis Phase* that we have presented in the previous section constitute the *Training Phase* of MaCAPA. The *Training Phase* has to be performed one time only per architecture. Since it is decoupled from the compilation of programs during the *Application Phase* of MaCAPA, the compile time is not increased by training. In particular, our ML techniques aim at achieving four goals. We aim at learning loop iteration counts and learning recursion frequencies to predict the communication amount between tasks if communication arises within loops that do not have statically determinable loop bounds or within recursive functions. For example, at parallel stencil computations (usually performed via loops over the grid), the halo elements of the stencil are communication points. Likewise, parallel branch and bound algorithms over recursive structures like a tree (whose natural implementation is to use a recursive function) have to communicate local solutions to other parallel tasks. Furthermore, we aim at learning execution times of tasks to predict their computational cost, and learning the PE that executes a task most efficiently to treat heterogeneous architectures. We present our ML techniques to learn loop iteration counts in Subsection 4.3.1 and to learn recursion frequencies in Subsection 4.3.2 Then, we show our ML techniques to learn execution times in Subsection 4.3.3 and, based on this, to learn the best performing PE in Subsection 4.3.4.

4.3.1 Learning Iteration Counts of Loops

With this thesis we aim at, among other things, reducing the communication overhead between parallelly executable tasks at run-time. Since predicting the overhead involves estimates for the communication amount, predictions for loop iteration counts are required in case of communication that arises within loops. Our approach for learning loop iteration counts automatically generates classifiers that relate static code features to dynamic loop behavior, thus providing the compiler with intelligence. In particular, we employ the *Random Forest* learning technique proposed by Breiman [Bre01], which is a supervised learning technique based on classification trees (see Subsection 2.3.1). For classification learning, a loop is described by a d_l-dimensional feature vector $\mathbf{x}_l \in \mathbf{X}_l$ and its loop iteration count is categorized concerning a set of given classes $\mathbf{C}_l = \{C_{l_i} \mid i = 1, \ldots, k_l\}$. The aim is to build a statistical model that best explains the relationship from features to classes for the training data. As we have introduced in Subsection 4.2.1, we consider $d_l = 114$ features. To classify the loop iteration count, we use the truncated decadic logarithm. In other words, the number of digits within the iteration count determines the class. Note that we are completely free in choosing the classification. For example, we can use three classes with the meaning of a small, moderate, and huge loop iteration count, respectively. Instead we have chosen a more fine grained classification. However, due to the logarithmic scale, the

highest class still comprises a great range of iteration counts. Nevertheless, a loop predicted to be in this class by all means contributes to massive communication overhead, no matter where the iteration count is located within that range. Given the training data, which we derive during the *Analysis Phase* of MaCAPA, classification learning yields a predictor function

$$pred_l : \mathbf{X}_l \rightarrow \mathbf{C}_l \ .$$

In the *Application Phase* of MaCAPA, the function $pred_l$ can be applied to feature vectors of unseen loops to classify their iteration count. To that end, we simply have to extract the code features of each loop via fast and highly scalable static analysis. Our approach thus automatically generates heuristics that efficiently provide the compiler with knowledge of run-time behavior, namely to predict loop iteration counts. Since the heuristics are automatically generated and incorporated into the compiler without the need for user intervention, our approach preserves an automatic, continuous compilation flow. Note that the generated heuristics can also be combined with static program analyses. When the analysis knows the results to be exact, this result can be used, otherwise, the heuristics are consulted. Thereby, our approach combines the benefits of static analyses and profiling without taking their disadvantages.

Additionally to our primary goal of reducing communication overhead, our learned classifiers for loop iteration counts can be used by other optimizations within the compiler framework. Often, program execution spends most of the time in a small fraction of code, a characteristic known as the "90-10 rule" – 90% of the execution time comes from 10% of the code. About 85% of those regions are inner loops, while the remaining 15% are functions [SNV+03]. These code fractions, called *hot spots* of a program, are regions where optimization would be most beneficial to improve the run-time performance of programs. The study of Villarreal et al. [VLCV01] also shows that programs spend on average about 70% of their execution time in loops. They additionally observe that many loops iterate only once or just a few times, which is also confirmed by our observation. This implies that only certain loops are actual hot spots of the program because not all loops significantly contribute to the execution time. Hence, our approach for learning iteration counts of loops that do not have statically determinable loop bounds can also be used for identifying actual hot spots of a program. We already have published our first approach using the classification tree learning technique [BFOS84] and have presented experimental results in [TG10]. In [TG13b], we have presented our improved learning method. Using *Random Forest* as learning technique, our approach yields notable improvements of experimental results compared against our previous results. Learning loop iteration counts enables us to precisely predict the expected number of iterations as is demonstrated by our experimental results (see Subsection 7.2.1), showing that our predictions deviate from the actual classified run-time behavior by only 0.72 classes on average.

4.3.2 Learning Recursion Frequencies of Functions

Regardless of the programming language, there are tasks such as string and tree traversal during parsing and interpreting source code, media decoding and encoding, or analyses in the biological domain for which the most natural implementation uses a recursive algorithm. Though static analyses can predict its execution frequencies, the estimates are imprecise because recursion frequency is rarely predictable using static techniques. Consequently, it is necessary to automatically incorporate knowledge of dynamic program behavior into the compiler to enable more precise predictions without the need for manual annotations or profiling to preserve an automatic, continuous, and efficient compilation flow. Moreover, the technique should be highly scalable to facilitate the integration in industrial-strength compiler environments used for compilation of real-world applications. We tackle this problem of incorporating information about recursion frequencies of functions again with *Random Forest* learning [Bre01]. Because we use again a concise representation for classification learning (decision trees), the obtained predictor can be efficiently implemented (the resulting code consists of nested `if-else` statements). To learn recursion frequencies, a function is presented by a d_f-dimensional feature vector $\mathbf{x}_f \in \mathbf{X}_f$. We categorize its recursion frequency concerning a set of given classes $C_f = \{C_{f_i} \mid i = 1, \ldots, k_f\}$. Given the training data derived during the *Analysis Phase* of MaCAPA, our learning algorithm yields a predictor function

$$pred_f : \mathbf{X}_f \rightarrow \mathbf{C}_f$$

that can be applied to feature vectors of unseen recursive functions. We take advantage of this function $pred_f$ as automatic heuristics for predicting the recursion frequency of functions at compile time of programs. During the *Application Phase* of MaCAPA, we use these heuristics for deriving the communication overhead between tasks (in case of communication that arises within recursive functions).

We use the truncated decadic logarithm of the recursion frequency as classification and we consider $d_f = 19$ features as we have introduced in Subsection 4.2.1. Again, we are completely free in choosing the classification and could use three classes with the meaning of a small, moderate, and huge recursion frequency, respectively. Instead we have chosen a more fine grained classification. However, the highest class still comprises a great range of recursion frequencies. But we argue that a function predicted to be in this class by all means contributes to massive communication overhead, no matter where the recursion frequency is located within that range. To obtain a prediction during the *Application Phase* of MaCAPA, we simply have to extract the code features via fast and highly scalable static analysis. Due to the high scalability of the resulting heuristics, the additional compile-time overhead is negligible and lower than those of conservative program analyses. Our approach for learning recursion frequencies, which we have presented in [TG12b], thus preserves an efficient compilation flow.

4.3.3 Learning Execution Times of Tasks

The aim of this thesis is to improve the mapping of concurrent applications to parallel architectures, which requires predictions for execution times of parallelly executable tasks. Though purely static analyses also can estimate runtime behavior, the estimates are imprecise because their analytical results must consider all possible cases how a program can behave. We address this problem by use of ML techniques. As we have introduced in Subsection 4.2.1, we use equivalence classes of machine code instructions with respect to similar behavior as features. Since we consider 13 classes as features, we have d_t-dimensional feature vectors $\mathbf{x}_t \in \mathbf{X}_t$ with $d_t = 13$. We assume that there exists a linear relationship between the amount of (classified) machine code instructions to be executed by the task and its execution time. Therefore, we consider *linear regression modeling*, for which several learning algorithms exist (see Subsection 2.3.2). Each learning algorithm yields the hypothesis h^*, i. e., a certain linear model $g^*(\bullet)$, that approximates the linear relationship as closely as possible. Given a feature vector $\mathbf{x}_t$ of a task, the output of $g^*(\mathbf{x}_t)$ is a numerical value in floating point representation that we use as prediction for the execution time of the task. Consequently, a learning algorithm yields a predictor function for the execution time

$$\begin{aligned} &pred_t : \mathbf{X}_t \to \mathbb{R} \\ &pred_t(\mathbf{x}_t) = g^*(\mathbf{x}_t) \ . \end{aligned}$$

We have evaluated the different learning algorithms for regression modeling that we have introduced in Subsection 2.3.2. Additionally, we have adapted the *Predicting Query Run-time 2* (*PQR2*) technique of Matsunaga and Fortes [MF10] that is based on a decision tree (see Page 63 for the original approach). We exclusively use static code features of applications as input and we also have employed *Naïve Bayes* as possible classifier at inner nodes. Furthermore, instead of using only k-NN classifiers with $k = 1$ and $k = 3$ at inner nodes, we use a ten range grid search for the best k between $k = 1$ and $k = 10$. Our evaluation of all algorithms (see Section 7.2.3) shows that it is advantageous to apply our adaption of the PQR2 algorithm for statically predicting execution times. We therefore propose to use it for establishing the predictor function $pred_t$, which can then be applied to feature vectors of unseen tasks during the *Application Phase* of MaCAPA. We already have published our approach and have presented experimental results in [TG13a].

4.3.4 Learning the Best Performing PE

We propose two variants for learning the PE of a heterogeneous architecture that executes a task most efficiently, which we call the *basic* and the *extended* variant. Assume a target architecture with n different PEs. The *basic* variant establishes an ML predictor for each different PE of the target architecture with our approach for execution time learning as described in the previous

subsection. That is, n distinct ML models have to be learned and a feature vector of a task is given by $\mathbf{x}_p \in \mathbf{X}_p$ with $\mathbf{X}_p = \mathbf{X}_t$. This yields n predictor functions $pred_{t_p} : \mathbf{X}_p \to \mathbb{R}$ $(p = 1, \ldots, n)$ and we obtain n predictions for the execution time on a particular PE from these predictor functions. We thus can determine the PE that executes a task most efficiently by choosing the PE with the lowest execution time prediction. Hence, the basic variant establishes a predictor function $pred_p$ for the best performing PE that labels one of the n different PEs. More formally, this yields the predictor function

$$pred_p : \mathbf{X}_p \to \{p \in \mathbb{N} \,|\, p = 1, \ldots, n\}$$
$$pred_p(\mathbf{x}_p) = \arg \min_{p \in \{1,\ldots,n\}} pred_{t_p}(\mathbf{x}_p)$$

where arg min returns the argument p that minimizes $pred_{t_p}(\mathbf{x}_p)$. We state that execution times will always depend on executed instructions (which we categorize according to similar behavior). Therefore, we conclude that our scheme for learning execution times will also generalize to other architectures and the basic variant will thus predict the best performing PE with the same accuracy. In Section 7.2.4, we show that we are able to precisely predict the PE that executes a task most efficiently with our basic variant. In [TG13a], we have presented this basic variant for learning the best performing PE.

The *extended* variant establishes a single ML model, where the feature vector for learning execution times is extended with further static code features and architectural features. As we have described in Subsection 4.2.1, the feature vector for learning execution times with $d_t = 13$ dimensions can be extended by 33 features (22 architectural features and 11 further static code features), which leads to 46-dimensional feature vectors $\mathbf{x}_p \in \mathbf{X}_p$ for learning the best performing PE. Depending on the chosen ML technique, the learning goal can be either classification or regression. Using regression modeling where the execution time of tasks is used as correct outcome for the observations, the learning algorithm yields a certain model $g^*(\bullet)$ that approximates the linear relationship between features and execution time as closely as possible. From this model, we can obtain the execution time predictions for a task on n different PEs by holding all the static code features constant and varying the 22 architectural features according to each PE. This automatically yields the best performing PE by choosing the PE with the lowest execution time prediction. More formally, assume the set of n different feature vectors derived from the static code features of a task (by varying the architectural features) is represented by $\mathbf{X}_n$ and a labeling function pe maps the feature vector $\mathbf{x}_p$ according to its contained architectural features to the corresponding PE[6]. Then, the predictor function $pred_p$ is given by

$$pred_p : \mathbf{X}_p \to \{p \in \mathbb{N} \,|\, p = 1, \ldots, n\}$$
$$pred_p(\mathbf{x}_p) = pe(\arg \min_{\mathbf{x}_p \in \mathbf{X}_n} g^*(\mathbf{x}_p))$$

[6]The mapping function pe is automatically obtained during the *Analysis Phase* of MaCAPA by recording which PE has led to which architectural features.

where arg min returns the argument $\mathbf{x}_p$ that minimizes $g^*(\mathbf{x}_p)$. For this proposed variant, we have no assessment of quality because it is not present in the instantiation of MaCAPA for improving the mapping of MPI programs to processor networks. However, we expect that our adaption of the PQR2 algorithm would also yield accurate results because most of the additional 33 features are binary features or categorical features with a few number of possible values, for which decision tree based learning algorithms are appropriate. At inner nodes of the PQR2 tree, where classifiers determine the threshold value to split the input space in two, these binary and categorical variables are suitable and a learning algorithm works rather well in general (if a relationship exists). We state that there exists a relationship, not only between categorized instructions and the execution time but also between the target architecture, which we represent by 22 features, and the execution time. Since the same applies to classification tree learning, we also expect accurate results when the goal of our extended variant is classification. Using classification for the extended variant with n classes representing the n different PEs, the obtained model $g^*(\bullet)$ directly returns the PE that executes a task most efficiently given the extended feature vector. Consequently, the classification algorithm yields the predictor function

$$\begin{aligned} pred_p &: \mathbf{X}_p \rightarrow \{p \in \mathbb{N} \,|\, p = 1, \ldots, n\} \\ pred_p(\mathbf{x}_p) &= g^*(\mathbf{x}_p) \quad . \end{aligned}$$

We propose to use both the basic and the extended variant for learning. Our model using the extended variant directly yields the PE that executes a task most efficiently based on code features and architectural features and can be used for allocating tasks to the best performing PE. If the best performing PE is not available (because other tasks are yet allocated there), our basic variant is used for determining the second best, the third best if the second is also not available, and so on.

In this section, we have shown how ML can be used to build predictors of regarded run-time behavior during the *Machine Learning Phase* of our proposed framework MaCAPA. In particular, we have described our approaches for learning iteration counts of loops, recursion frequencies of functions, execution times of tasks, and the PE that executes a task most efficiently. Once the machine learned models are established, they can be used from then on without modification. This eliminates the need for profiling at the *Application Phase* of MaCAPA, nonetheless incorporating predictions for anticipated run-time behavior into the compilation flow. In the next section, we describe the *Application Phase* of MaCAPA.

4.4 Mapping Applications to Parallel Architectures

With this thesis, we aim at improving the mapping of concurrent applications to parallel architectures. A vital part of this mapping is the allocation of parallelly executable tasks of an application to PEs of the target architecture,

which requires predictions for the execution time of tasks and for the communication between tasks. In the previous sections, we presented the one-off *Training Phase* per architecture of our proposed framework MaCAPA, in which we automatically build predictors for the required run-time behavior. In this section, we present the *Application Phase* of MaCAPA, where applications are compiled and the mapping of tasks to PEs is derived based on our established predictors. Basically, we first analyze parallelly executable tasks of applications – instructions to be executed by each task as well as the communication and interdependencies between tasks. Then, a cost model that is constructed based on our task analysis employs the established behavior predictors from the *Machine Learning Phase* to rate the gain of alternative task mappings. On this basis, we map the tasks power- and communication-aware to the PEs. In the following, we introduce our static task analysis in Subsection 4.4.1 and we show how the learned predictors are used to build a precise cost model in Subsection 4.4.2. Afterwards, we present our approach for improving the mapping in Subsection 4.4.3.

4.4.1 Static Analysis of Concurrent Tasks

A major challenge of mapping concurrent applications to parallel architectures is to analyze the parallelly executable tasks of applications. This involves to determine which task executes which part of the program, among which tasks arises communication, and the interdependencies between tasks. We have developed automatic analyses that statically yield close approximations for this run-time information. In this section, we present our analyses. The analysis of instructions that may be executed by a task is a prerequisite for predicting the execution times of tasks and for identifying the communication and interdependencies between tasks. With our analysis of communication partners, we are able to minimize the communication overheads that have emerged as a major performance limitation of concurrent applications. Our analysis of interdependencies considers the synchronization between tasks, which helps to determine precedence constraints on the execution of tasks. Among others, we use this information together with the predicted execution times of tasks to identify long running tasks on which other tasks depend. A delay of the execution of these tasks also delays the execution of dependent tasks, which would degrade the run-time performance of the application. Our analysis of interdependencies enables an improved scheduling because we favor the execution of these tasks. Based on the information given by our analyses, we derive a precise cost model for the computational cost of tasks and for the communication cost between tasks regarding alternative allocations.

As input to our framework MaCAPA, we take the source code of programs. The front end of the compiler then transforms the source code into an IR on which our analyses work. We assume that a program is written in a parallel programming language, e. g., UPC, which realizes the *Distributed Shared Memory Model*, MPI, which realizes the *Message Passing Model*, and threaded C or OpenMP, both realizing the *Shared Memory Model*. That is, they have an explicit parallel structure in common and parallelly executable tasks are

given. We therefore do not have to parallelize a program. As we have seen in Subsection 4.2.2, tasks are implemented by functions for the programming languages mentioned above (compare Figure 4.3). However, tasks may be expressed in different ways depending on the programming language and the realized task-level parallel programming model. For programs that execute in SPMD mode, such as programs written in UPC and MPI, tasks are the whole program (i. e., the `main` function) and *environmental variables* define which part of it is executed by which task. Using threaded C or OpenMP, tasks are specified via library calls[7] and execute the corresponding function. Hence, we first have to analyze the *Control Flow Graph* (*CFG*) of functions that implement tasks, from which we obtain the instructions that may be executed by each task. Afterwards, our analysis identifies the communication between tasks. For shared (or distributed shared) parallel programming models, tasks are best created as threads since they execute in one (logically shared) address space. In this case, information exchange can be performed by direct read or write accesses to shared variables. The message passing model where the memory is distributed forces the creation of tasks as processes that execute in their own local address space. This requires explicit communication calls for information exchange between tasks. Moreover, tasks may have further interdependencies, which we analyze. They can, e. g., synchronize with each other, either by synchronous communication or by explicit directives that constitute barriers. In the following, we describe our analyses in more detail.

Analysis of Executed Instructions

The basis of the *Application Phase* of MaCAPA is our analysis of instructions that may be executed by a task. It is vital for analyzing the communication and interdependencies between tasks and for predicting execution times of tasks. The basic idea of our analysis is to separate run-time behavior that is specific for a task from data dependent run-time behavior. Since our analysis works on the IR, the compiler has yet processed the source code written in a certain programming language. For each programming language (or library extending a given language), it is given how tasks are created, e. g., by the whole program or via library calls. Therefore, we can automatically derive which functions represent tasks. As we have discussed in Subsection 4.2.2, the CFGs of tasks are always given.

Our approach to achieve this is as follows. First, we analyze the CFGs of tasks to determine the instructions that each task can execute. Based on this, we derive a static, context-insensitive CG for each task. As introduced in Subsection 2.1.2, a CG is a multigraph that captures every function call relationship between a *caller* and the *callee* in the application. The CG cg of the whole application is represented by $cg = (\mathcal{F}, \mathcal{C})$ where $\mathcal{F}$ is the set of functions defined within the application and the multiset $\mathcal{C}$ constitutes the

[7] For OpenMP, the preprocessor directive `#pragma omp task` yields a library call to fork a task. This task executes a wrapper function that contains the corresponding code block.

function call relationship. Formally, we define the CFGs and a CG of a task as follows.

Definition 4.1 (Control flow graph of functions)**.** *A function $f \in \mathcal{F}$ is given by a CFG cfg_f with $cfg_f = (B_f, E_f)$, B_f is the set of BBs in the IR of the function body and edges $e \in E_f$ represent control flow. Typically, the compiler adds information to these edges, such as the execution probability or whether it is a back edge. The set of all CFGs is hence defined by*

$$CFG := \bigcup_{f \in \mathcal{F}} cfg_f \ .$$

Definition 4.2 (Call graph of tasks)**.** *A CG cg_t of a task t is given by $cg_t = (F_t, C_t)$ where $F_t \subseteq \mathcal{F}$ is the subset of functions defined by the application that are called from the task t (directly or indirectly via calls by callees of t) and an element in $C_t \subseteq \mathcal{C}$ represents that a function $f \in F_t$ calls a function $g \in F_t$. Each function $f \in F_t$ is represented by a CFG $cfg_f \in CFG$.*

Note that the total number of tasks that execute a concurrent application does not have to be statically given. For example, the number of threads to execute a UPC program can be specified at compile time or at run-time. In both cases, the global variable `THREADS` holds this number (for the former it is the constant known to the compiler). The same applies to OpenMP, where the number of tasks can dynamically be set by a library routine for parallel regions that do not statically specify this number. At run-time, the actual number can be obtained with a library call. Using MPI, the number of processes that are supposed to execute the binary must be specified at startup and can also be obtained via a library call. We call the variable that holds the number of tasks at run-time an *environmental variable*. For threaded C, the threads can be created within loops whose iteration count depends on run-time values. If the total number of tasks is not statically determinable we use a symbolic upper bound and otherwise the actual number.

Both the CFG and the CG together represent the complete possible behavior of a task. Additionally, we consider further environmental variables that may constrain the possible run-time behavior. Tasks can have a unique value assigned to environmental variables for identifying them. From now on, we call this value the *ID* of the task. Examples for IDs are the so called *rank* of MPI processes, the value of the global variable `MYTHREAD` of UPC threads, or local values given as argument to the function (compare Figure 4.3). If conditional paths within the CFG depend on these IDs, only the task holding the corresponding value will execute the paths. Environmental variables can be assigned to other variables or can be passed as an argument to other functions (where they may be assigned to variables or passed to functions as well). To determine all constraints, we provide a *Least Fixed-point Algorithm* (see Algorithm 4.4). Our algorithm records the data flow of environmental variables and analyzes the paths in the CFG that depend on the IDs of tasks. This determines the behavior of tasks as closely as possible, which leads to the following definition.

Definition 4.3 (ID dependent behavior of tasks)**.** *The behavior of a task t is as closely as possible represented by $T_t := (cfg_t^{ID}, cg_t^{ID})$, where cfg_t^{ID} and cg_t^{ID} are the result of Algorithm 4.4. The set of representations of all tasks of an application is hence given with $\mathcal{T} := \{T_i \,|\, i = 1, \ldots, n\}$, where n is the number of tasks within the application.*

With each CFG cfg_t of a task t, we associate a set of environmental variables that affect the behavior of this task. Initially, the variable that holds the ID of the task is in this set. We access this set via `env(`cfg_t`)` and we provide a function `DUchain`. This function analyzes the uses of defined variables in `env(`cfg_t`)`, i. e., to which other variables they are assigned, and updates the set accordingly. Additionally, sets are constructed for every BB of the CFG to record which variables hold task dependent values at which BB. We use this information to precisely analyze whether conditional paths in the CFG will be taken depending on the ID of a task. We call the edges of a CFG that correspond to conditional paths *conditional edges*. Each conditional branch in a CFG cfg_t defines guards for the corresponding paths, which determine whether they are taken or not. Based on these guards, we define a predicate `pred` on conditional edges that considers task dependent behavior. It evaluates to true if the guard does not depend on an environmental variable, i. e., on an element $var \in$ `env(`cfg_t`)`, or the guard depends on an environmental variable and the comparison against the value of *var* evaluates to true. Intuitively, if the guard for a path checks whether the value of an environmental variable is in a certain range and the ID of a task fulfills this condition such that the path is taken, the predicate evaluates to true. If the condition is not fulfilled, i. e., the path can only be taken by other tasks, the predicate evaluates to false. When the condition does not check an environmental variable, the predicate evaluates to true because the path will be taken or not depending on other run-time values but not depending on the ID of the task. For a BB *bb* in a CFG, the function `numPredecessors` returns the number of edges that target *bb*. Similarly, $\texttt{numCalls}_f$ returns for a function f the number of edges in a CG that target f.

The algorithm works as follows. First, we initialize the results with the input CFG and CG of the task (Line 1) and the working set $CFGs$ with its CFG (Line 2). When the working set gets empty, the fixed-point of our algorithm is reached. At Line 5, we update the set of environmental variables. This update associates with each BB a set of variables that hold environmental values at this point. Our algorithm then checks for each conditional edge whether the corresponding path in the CFG cannot be taken due to the current task ID (Lines 6 to 7). In this case, the path is removed from the CFG by our function `removePath` (Line 8). The function `removePath` therefore deletes the conditional edge (Line 17). Additionally, the function considers if the target BB of the edge cannot be reached (Line 18). If so, the BB is deleted from the CFG, calls to other functions within this BB are deleted from the CG because they cannot be called, and the outgoing paths of the BB are transitively removed (Lines 19 to 25). Note that the unique *entry BB* of each CFG is considered to always have at least one ingoing (pseudo) edge in the CG to cope with loops. Otherwise, for a path to be deleted in a CFG that targets the entry BB, the transitive removal would delete the whole CFG. Similarly,

Algorithm 4.4: Determine behavior of tasks

Input : CFG cfg_t and CG cg_t of a task t, CFGs cfg_i of functions
Result : ID dependent CFG cfg_t^{ID} and CG cg_t^{ID}

1 $cfg_t^{ID} = cfg_t$; $cg_t^{ID} = cg_t$;
2 $CFGs = \{cfg_t^{ID}\}$; $(F,C) = cg_t^{ID}$;
3 **while** $CFGs \neq \emptyset$ **do**
4 $(B,E) = cfg_f \in CFGs$; $CFGs = CFGs \setminus \{cfg_f\}$;
5 `DUchain(env(`cfg_f`))`;
6 **foreach** conditional edge $e : bb_s \to bb_t$ in E **do**
7 **if** $\neg$`pred(`e`)` **then**
8 `removePath(`cfg_f, e`);` *(* path cannot be taken *)*
9 **foreach** edge $e : f \to g$ in C **do**
10 **foreach** variable $var \in$ `env(`cfg_f`)` **do**
11 **if** var is argument to g as parameter p_i **then**
12 **if** $p_i \notin$ `env(`cfg_g`)` **then**
13 `env(`cfg_g`)` $=$ `env(`cfg_g`)` $\cup \{p_i\}$; *(* record data flow *)*
14 $CFGs = CFGs \cup \{cfg_g\}$;

15 **function** `removePath(`CFG cfg_f, edge $e : bb_s \to bb_t$`)`
16 $(B,E) = cfg_f$;
17 $cfg_f = (B, E \setminus \{e\})$;
18 **if** `numPredecessors(`bb_t`)` $==$ 0 **then**
19 $cfg_f = (B \setminus \{bb_t\}, E)$;
20 **foreach** function call in bb_t having edge $e : f \to g \in C$ **do**
21 $cg_t^{ID} = (F, C \setminus \{e\})$;
22 **if** `numCalls`$_f$`(`g`)` $==$ 0 **then**
23 $cg_t^{ID} = (F \setminus \{g\}, C)$;
24 **foreach** edge $e_i : bb_t \to bb_i$ in E **do**
25 `removePath(`cfg_f, e_i`);` *(* transitive removal *)*

the function f that represents a task is considered to always have at least one ingoing (pseudo) edge to cope with recursive function calls. Otherwise, a recursive call to be deleted that targets f would result in deleting the task itself from the CG (Line 23)[8]. Our algorithm also examines the data flow of environmental variables across function calls (Lines 9 to 10). If a function is called with an environmental variable as argument, the corresponding parameter should be included in the set of environmental variables of this function because conditional paths depending on the parameter then also depend on the ID of the task. To ensure that the fixed-point is eventually reached, the update that a parameter holds a task dependent value is only performed once

[8]The other functions that do not represent a task are not considered to have at least one ingoing edge. If recursive calls occur by these functions without being called from the task the functions can be deleted from the CG.

per parameter (Lines 12 and 13). In case of an update, the function must be analyzed again (Line 14) because this new information may result in removing further conditional paths than detected before.

Our proposed algorithm always reaches a least fixed-point. We start from the complete CFG $cfg_t = (B_t, E_t)$ and the static CG $cg_t = (F_t, C_t)$ of a task together with the CFGs $cfg_i = (B_i, E_i)$ of all other functions. All sets B_t, E_t, F_t, C_t, B_i, and E_i are finite and our algorithm does not insert new elements in these sets (instead, it only removes elements). Most important, no new function calls are inserted, i. e., the number of function calls remains finite. The algorithm proceeds only when it is first determined that an argument of a function call holds an environmental variable. Since the number of variables and the number of parameters of functions are finite, the least fixed-point is eventually reached. From our analysis as introduced above, we obtain the instructions that may be executed by each task. Afterwards, we analyze the communication between tasks, which we present in the following.

Analysis of Communication Volume

Based on our analysis of instructions that may be executed by a task, we analyze the communication between tasks to enable the minimization of communication overhead at run-time of applications. This involves to identify communication partners and the amount of communication. For this purpose, we analyze the ID dependent CFGs and CGs of tasks, which our analysis of executed instruction yields. The result of our analysis can be seen as a task communication graph, which leads to the following definition.

Definition 4.4 (Task communication graph)**.** *The communication between tasks is represented by a task communication graph*

$$TCG := (\mathcal{T}, \texttt{comm} \subseteq \mathcal{B} \times \mathcal{B} \times \mathbb{N})$$

where $\mathcal{T}$ is the set of the representations of all tasks as given by Definition 4.3, $\mathcal{B}$ is the universe of BBs, and `comm` *denotes the communication volume.*

For tasks that are realized as threads (having a shared address space), there is one global memory that can be accessed from all tasks and information exchange can be performed by direct read or write accesses to *shared* variables. In case of using shared variables and the value of a shared global variable is not present in a register, an access to a variable means to load its value from memory or from a cache (if existent), which is much faster. Sharing a cache will therefore speedup communication. Since, e. g., at most the L2, L3 and L4 data caches are usually shared between processor cores while the faster L1 cache is private to one core, executing tasks on the same core could reduce the communication overhead.

When tasks are realized by processes, each process executes in its own address space. Hence, the communication is explicit because they cannot use shared variables to communicate. In this case, our communication analysis

determines these explicit communication calls, which can be categorized according to the communication partners as *one-to-one*, *one-to-all* (also called *broadcast*), *all-to-one*, or *all-to-all*. We can automatically derive in which category a communication call falls since our compiler front end has yet processed the source code written in a certain programming language (where the category is known) and can annotate this information to the corresponding IR construct. The term communication in this context can be understood as an exchange of messages between tasks. To determine communication partners, the analysis of broadcast and all-to-all calls is straightforward. We visit the CFGs of the tasks[9] and if we record a broadcast or all-to-all call, we know that all tasks will send or receive messages at this point of execution. For the former category, all tasks will receive a message from the task whose CFG we are currently visiting and for the latter category, all tasks will exchange messages with all other tasks. Hence, both variants only differ in the number of partners. For tasks that communicate with each other, we create directed *communication edges* that connect the CFGs of the participating tasks.

Our analysis of one-to-one and all-to-one message passing calls must determine the single partner in order to connect the corresponding CFGs with communication edges. Explicit communication requires that the single partner must be given, e. g., identified by its ID. However, the value for identifying the partner of a message passing call does not need to be a constant in the IR of the application. For example, it can be a variable which holds a value that depends on the input data of the program. Our analysis considers the cases where a static analysis is able to determine the ID of the partner: when it is derived from a value by using basic arithmetic operations and we know that value. For example, the value can be a constant, the ID of the task that initiates the communication, the total number of tasks, a value (transitively) derived from these former values, or a special tag that connects a send operation with a receive operation. From these values, we identify the partner by tracking the used arithmetic operations. When the exact partner cannot be statically determined, we analyze possible partners and we create corresponding communication edges for all of them. With this over-approximation, we cover the whole communication behavior at run-time of applications since possible communication partners cannot be overlooked.

Tasks that are realized as threads can also exchange information via message passing. Hence, our communication analysis for processes as introduced above applies here as well. Additionally, our communication analysis involves to detect global variables that are accessed by more than one task, i. e., whether they are shared. For these global variables, we determine the set of tasks that can access their values at a certain point of execution. This set may contain more tasks that actually can communicate at run-time via the variable due to conditional paths depending on run-time values but this over-approximation

[9] We visit not only the CFG cfg_t^{ID} of a task t but also the CFGs of called functions in its CG cg_t^{ID} since communication can also occur there. In the following, we subsume these CFGs under the CFG of the task.

is statically unavoidable[10]. Then, we analyze in each set the possible pairs of tasks where the former writes a value that the latter reads and we create for each combination of such pairs in the sets corresponding communication edges. Consider, for example, a task t_1 that reads a value from a variable and writes another value to this variable and a task t_2 then performs the same. Afterwards a task t_3 reads from this variable. In this ordering, there is in fact only a communication between the pairs (t_1,t_2) and (t_2,t_3). Since we cannot statically determine the interleavings of executions at run-time, it could be also possible that t_2 performs the same before t_1 such that only (t_2,t_1) and (t_1,t_3) communicate. Therefore, we create edges for all combinations, which simply makes transitive edges explicit. If, for example, task t_2 does not write to this variable then we create only communication edges for the pairs (t_1,t_2) and (t_1,t_3) because we consider only possible reads after writes.

To minimize the communication overhead at run-time of applications, we also analyze the amount of each communication, which we call the communication *volume*. Then, we weight each communication edge with this communication volume. Since we also weight the transitive edges as discussed above with the volume, the actual amount of communication may be smaller when edges are created for pairs that do not directly communicate at run-time. However, this over-approximation cannot be statically avoided. For message passing communication, the volume to be passed is the size of the data type times the number of elements of that type, which must be given to the corresponding call. As with the ID of the communication partner, the value of the number of elements does not have to be a constant in the IR of the application. Again, our analysis considers all cases where a static analysis is able to determine this number. In the case where we cannot determine the number of elements, we assume a fixed amount as volume.

When shared global variables are used to communicate, their size is important to determine the amount of data that must be accessed from memory (if its value is not in a register). For a basic data type, we take its size as communication volume. If the accessed variable is a composite structure, we determine its *static size*, which is the sum of the data type sizes of its elements, and use it as communication volume. That is, when the structure contains for example pointers that constitute a recursive structure (such as linked lists for which we cannot statically determine the number of elements pointed to), we take the size of the pointer data type because only the reference to its first element is communicated. Likewise, for arrays we take the size of one element as communication volume. Note that we do not have to consider the *dynamic size* of composite structures, i. e., how many elements will be actually allocated at run-time of applications. Our communication analysis as described above records the communication volume for one access to shared global variables or for one message passing, respectively. The total number of elements of composite structures to be transferred during communication, e. g., via looping over array elements or via recursion over recursive structures, is determined

[10] A context-sensitive static analysis that precisely takes into account the constraints on execution order imposed by the synchronization statements (like message passing) in the program is impossible even for the simplest of analysis problems [Ram00].

with our cost model that inspects the surrounding IR and employs our corresponding ML predictor functions to predict the number of loop iterations and the recursion frequency. Thus, we capture predictions for the amount of data during communication in all cases.

In summary, our static communication analysis yields for each task its ID dependent CFG that is connected via communication edges to CFGs of other tasks at points of execution where communication arises. Each communication edge is weighted by the volume to be transferred, which enables to minimize the communication overhead at run-time of applications. The result of our analysis is the task communication graph as given by Definition 4.4. We additionally analyze further interdependencies between tasks, which we present in the following.

Analysis of Interdependencies

Our communication analysis as introduced above considers the identification of communication partners and the communication volume to be transferred at points of execution of tasks. Our analysis result so far is a task communication graph. Additionally to the communication overhead, the performance of applications can be limited by other interdependencies between tasks, which we analyze to improve the run-time performance. Communication can also be seen as an interdependency between tasks – the communication volume must be transferred before a task can access data that is computed by another task. Besides this enforced ordering to transfer data, communication may yield further interdependencies between tasks. They can additionally synchronize with each other, i. e., the communication partners will not continue to execute their computations until they rendezvous at the communication. This can be accomplished by employing approaches to synchronization that provide mutual exclusion for shared resources (so called *locks* or *mutexes*) or by explicit synchronous message passing. Locks and mutexes, which are predefined data types in several programming languages, are global variables that can be locked when a task is in a critical code region and afterwards unlocked using library calls. To analyze the synchronization, we employ our function `DUchain` as described in Subsection 4.4.1 on these global variables to record the locks and mutexes at each BB. As with the other shared variables that we detect with our communication analysis, we determine the set of tasks that access their values at a certain point of execution. Then, we create corresponding communication edges for each combination of pairs of tasks in the sets. For message passing, we can automatically derive which message passing calls are synchronous and which are not. Our analysis of interdependencies then tags the corresponding communication edges with the information whether synchronous or asynchronous communication is performed.

Synchronization can also be achieved independently from communication via directives that constitute barriers. Barriers constitute important means to enforce an ordering constraint on the execution and are present in several APIs for parallel programming models. A common scenario is that tasks access data

that must be initialized by only one task. A synchronization barrier can then be used to ensure that the data is initialized before any of the tasks is allowed to run and uses the data. They are also useful for phased computations, in which tasks executing the same code in parallel must all complete one phase before moving to the next. In each case, a barrier is an expensive synchronization mechanism since the semantics of barriers require the computation to wait for the *slowest* task to arrive before the rest can proceed. Therefore, it is vital to consider these barriers during task mapping. For a barrier, it can be specified which tasks participate. All specified tasks will block their computation on encountering the barrier during execution until the last participating task encounters the barrier.

With our ideas described so far, our analysis of concurrent tasks connects the CFGs of tasks with communication edges that are weighted with the communication volume and are tagged whether the communication is synchronous or not. To reflect synchronization that is independent from communication, we insert *synchronization points* in our task communication graph. We use these synchronization points to analyze phased behavior and to enhance our communication analysis. Remember that we create communication edges to all possible partners if we cannot statically determine the exact partner. Hence, synchronization points are helpful to reduce the set of possible partners since only tasks executing the same phase must be taken into account. To capture the execution before and after the synchronization, we split each of the participating tasks into two parts, called *subtasks*. We call this analysis result a *Synchronized Task Interaction Graph* (*STIG*). To formally model our analysis result, we define the functions

- $\texttt{entryBB} : CFG \to \mathcal{B}$ that returns the unique entry BB of a CFG,
- $\texttt{exitBB} : CFG \to \mathcal{B}$ that returns the unique exit BB of a CFG,
- $\texttt{BBofFunCall} : \mathcal{C} \to \mathcal{B}$ that returns for a given function call in a CG the BB of a CFG from which the function call originates,
- $\texttt{barrier} : \mathcal{B} \to \mathbb{N}$ that labels the BBs how man synchronization statements are within the BB, and
- $\texttt{split} : \mathcal{B} \to \mathcal{B} \times \mathcal{B}$ that splits a BB bb of a synchronization point (i. e., with $\texttt{barrier}(bb) > 0$) into the tuple (bb_{bef}, bb_{aft}) where the BB bb_{bef} holds the instructions before the barrier and bb_{aft} the ones after the barrier. The BBs bb_{bef} and bb_{aft} are tagged with `barrier`-labels that holds the number of remaining synchronization statements such that $\texttt{barrier}(bb_{bef}) + \texttt{barrier}(bb_{aft}) + 1 = \texttt{barrier}(bb)$.

Based on this, the *synchronization points*, the *subtasks*, and our *STIG* are defined in the following.

Definition 4.5 (Synchronization points)**.** *The synchronization points are given with the relation* $\texttt{sync} \subseteq \{sync, wait, new_{cond}, new_{uncond}\} \times \mathcal{P}(\mathcal{B}) \times \mathcal{P}(\mathcal{B})$*. It relates the split BBs of tasks with the kind of synchronization we consider (we discuss the kinds below). An element* $(k, B_{bef}, B_{aft}) \in \texttt{sync}$ *represents that a synchronization of kind* k *occurs.* B_{bef} *is a set of BBs where each element is the BB of (a CFG of) a different task that holds the instructions before the*

barrier and B_{aft} is a set of BBs where each element is the BB of a different task that holds the instructions after the barrier.

Definition 4.6 (Subtasks)**.** *The subtask t_s of a task t with given representation T_t as defined in Definition 4.3 is represented by $T_t^s := (cfg_t^s, cg_t^s)$, where $cfg_t^s = (B_t^s, E_t^s)$ and $cg_t^s = (F_t^s, C_t^s)$ with $B_t^s \neq \emptyset$ and*

- $B_t^s \subseteq B_t^{ID} \cup \{bb \,|\, \exists b \in B_t^{ID} : ((b, bb, b_{aft}) \in \texttt{split}) \vee ((b, b_{bef}, bb) \in \texttt{split})\}$ (i)
  $\wedge (\forall\, bb \in B_t^s : (\mathsf{barrier}(bb) = 0))$ (ii)
  $\wedge (bb = \mathsf{entryBB}(cfg_t^s) \iff (bb = \mathsf{entryBB}(cfg_t^{ID})$ (iii)
  $\vee\, (bb \neq \mathsf{entryBB}(cfg_t^{ID}) \wedge \exists (b, bb_{bef}, bb_{aft}) \in \mathsf{split} : bb = bb_{aft})))$
  $\wedge (bb = \mathsf{exitBB}(cfg_t^s) \iff (bb = \mathsf{exitBB}(cfg_t^{ID})$ (iv)
  $\vee\, (bb \neq \mathsf{exitBB}(cfg_t^{ID}) \wedge \exists (b, bb_{bef}, bb_{aft}) \in \mathsf{split} : bb = bb_{bef})))$
  $\wedge (\neg (bb = \mathsf{entryBB}(cfg_t^s) \vee bb = \mathsf{exitBB}(cfg_t^s)) \implies$ (v)
  $(\nexists (b, bb_{bef}, bb_{aft}) \in \mathsf{split} : ((bb = bb_{bef}) \vee (bb = bb_{aft})))))$

  That is,

  (i) *the BBs of a subtask are a subset of the union of the BBs of the whole task and the BBs that are created with the function* `split` *on a BB of the whole task,*

  (ii) *and for all BBs of the subtask holds: there is no synchronization statement within the BB because it would have been split otherwise,*

  (iii) *it is the unique entry BB if it is the entry BB of the whole task (i. e., the subtask is the first part of the task) or it is a split BB with the instructions after the synchronization,*

  (iv) *it is the unique exit BB if it is the exit BB of the whole task (i. e., the subtask is the last part of the task) or it is a split BB with the instructions before the synchronization, and*

  (v) *the case that it is not the entry nor the exit BB implies that it is not a split BB (since another subtask would start or end there).*
- $E_t^s \subseteq E_t^{ID} \wedge ((b_s, b_t) \in E_t^s \implies ((b_s \in B_t^s) \wedge (b_t \in B_t^s)))$
- $F_t^s = F_t^{ID}$
- $C_t^s = \{c \in C_t^{ID} \,|\, \exists n \geq 1 : (\mathsf{BBofFunCall}(c_1) \in B_t^s) \wedge (c_n = c)$
  $\wedge\, (\forall i = 1, \ldots, n-1 : (c_i \in C_t^{ID} \wedge (c_i = (f, g) \implies c_{i+1} = (g, h))))\}$
  That is, (sequences of) function calls must originate within a BB of the subtask.

The set of representations of all subtasks of an application is hence given with $\mathcal{T}_s := \{T_i^s \,|\, i = 1, \ldots, n_s\}$, where n_s is the number of subtasks within the application.

Definition 4.7 (Synchronized Task Interaction Graph)**.** *The STIG is a tuple*

$$STIG := (\mathcal{T}_s, \texttt{comm}_s, \texttt{sync})$$

where

- *$\mathcal{T}_s$ is the set of representations of all subtasks as given by Definition 4.6,*

- $\texttt{comm}_s \subseteq \mathcal{B} \times \mathcal{B} \times \mathbb{N} \times \{sync, async\}$ *denotes the communication volume that is determined with the function* `comm` *as given by Definition 4.4 together with the information whether the communication is synchronous or asynchronous, and*
- *the relation* `sync` *represents synchronization between (BBs of) subtasks as given by Definition 4.5.*

Our analysis of barriers is performed on the ID dependent CFGs of tasks. We insert synchronization points in our task communication graph only when the barrier is in a BB b that post-dominates the entry BB of the CFG, which is the case if all paths between the entry BB and the exit BB contain b. That is, the task must not encounter the barrier conditionally depending on run-time values because we would not be able to determine execution phases in this case[11]. Note that we do not require a barrier to be encountered from all tasks for inserting synchronization points. Since we use the ID dependent CFGs of tasks, the task-dependent conditional paths (which cannot be taken) are removed. Hence, BBs contained in task dependent paths also post-dominate the entry BB if the BB with the condition post-dominates the entry BB. We thus also create synchronization points when a subset and not all tasks rendezvous at the barrier. This enables us to determine phases of program execution when a certain number of tasks participate at synchronization points. The phases are given by the dominance order of the BBs with barriers.

At synchronization points, we split each of the participating tasks into two parts, called subtasks, that represent the execution before and after the barrier. The executed instructions before the barrier are represented by all paths from the entry BB of the CFG to this point, i. e., to this BB (including the instructions in this BB before the barrier directive). The instructions after the synchronization barrier are those that are reachable from this BB in the CFG (including the instructions in this BB after the barrier directive). We analyze these subtasks and we create *synchronization edges* between all subtasks before the barrier with the synchronization point and between the synchronization point and all subtasks after the barrier. An example for the splitting of CFGs is given in Figure 4.4. BBs are shown as boxes, control flow edges as arcs, the synchronization barrier is marked orange, and communication is represented as a red line. In this example, tasks T_1 and T_2 first execute a phase where they may perform computations within a loop before they synchronize. Then, they may communicate with each other after the barrier. A more abstract view on the result of our synchronization analysis is depicted in Figure 4.5. Here, details of CFGs are replaced by hexagons. The labels of hexagons represent the task ID (before the dot) together with a number that denotes the execution part of the task (after the dot). Our created communication edge is shown as a dotted red arrow.

[11] Consider, for example, two barriers A and B with different partners that are in different conditional paths. Then we are not able to statically determine which path is executed, i. e., which partners encounter a barrier, when the condition depends on run-time values.

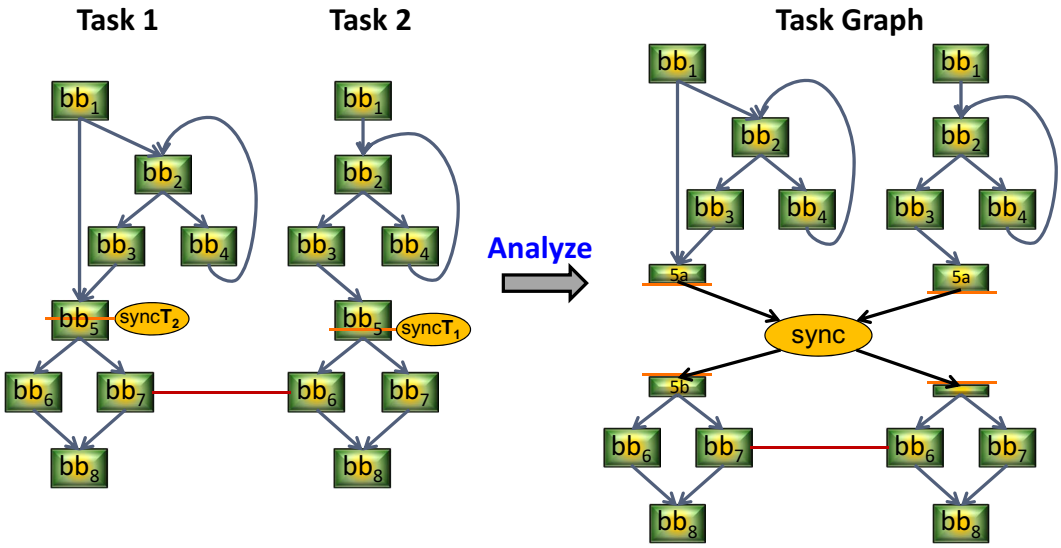


Figure 4.4: Analysis of synchronization between tasks

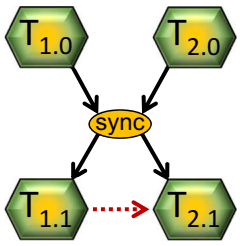


Fig. 4.5: Synchronization: abstract view

Special barriers are directives that require that a task must wait until another task has *finished* its execution. We call a barrier of this kind a *wait barrier*. This category of barriers allows us to reason about the number of tasks that are alive at this point of execution since the number of executing tasks decreases (by at least one) after such a barrier. Furthermore, this category is important for scheduling because a delayed execution of a task when another task waits for its completion extends the execution time of the waiting task. We therefore take all of these barriers into account, not only those occurring in BBs that post-dominate the entry BB. That is, we allow that wait barriers can be conditionally executed depending on run-time values. Hence, we must also consider that they can be performed within a loop. Since we are in general not able to statically determine which path will be taken at run-time, we assume that the path with the barrier will be taken. In terms of task scheduling our assumption means to optimize for the worst case scenario. Consider the execution of two tasks where one conditionally waits for the other to finish its execution. Instead of being able to entirely execute in parallel until completion, which is the case when the corresponding path is not taken, our scenario requires to wait for finishing. This may cause a waiting time and furthermore results in executing the waiting subtask after the barrier not in parallel with the other task, which may extend the total execution time.

We handle the three types of a wait barrier that are possible: a task has to wait for a certain task, for any task, or for all tasks, where waiting for any task means to wait for the first task that has finished its execution. When waiting for a certain task, the task must be given by its ID or by a handle that was returned at task creation. If we can statically determine the certain task or while waiting for all tasks, we distinguish two cases. When the wait barrier is in a post-dominator of the entry BB or within an unconditional path in a

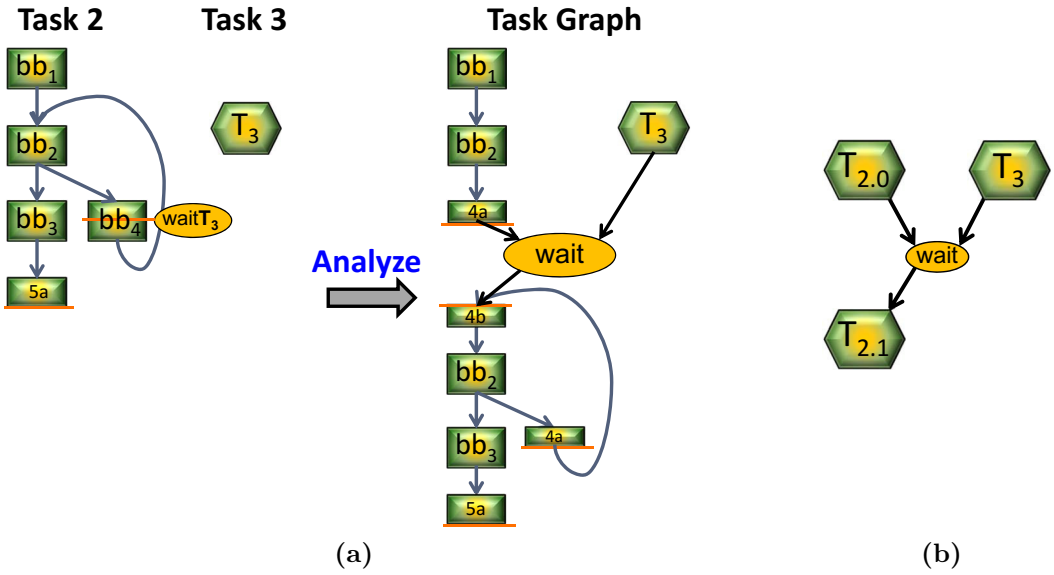


Figure 4.6: Analysis of wait barriers: **(a)** concrete and **(b)** abstract view

loop where the loop header is a post-dominator of the entry BB, we create a new synchronization point where the corresponding tasks end. Otherwise, we do not create a new synchronization point because we would not be able to establish an ordering between conditional synchronization points. Instead, we handle this case with modeling that the tasks end at the next synchronization point.

Consider again the example given in Figure 4.4 and assume that task T_2 has a wait barrier in BB bb_4 for requiring that a task T_3 has finished its execution. Because the loop header post-dominates the entry BB, we create a new synchronization point and we split the task T_2 at BB bb_4 (see Figure 4.6a that shows the subtask of T_2 before the synchronization with task T_1). Again, we connect the subtask of T_2 and task T_3 with synchronization edges. The subtask of T_2 before the barrier is solely the path from BB bb_1 to BB bb_4 (and not the path from BB bb_2 to bb_{5a}) since we assume that the path with the barrier is taken. Note that we do not generate machine code from our model because the semantics of the program is changed if the path with the barrier actually will not be taken at run-time. Instead, we generate machine code from the IR of the application and we exclusively use our model to analyze overheads and constraints for the execution. Any task that actually does not take the path with the barrier at run-time can therefore continuing execution without waiting for other tasks.

If the task T_1 shown in Figure 4.4 had a wait barrier in BB bb_4, we would not create a new synchronization point because the loop header BB bb_2 does not post-dominate the entry BB and the task T_3 would be modeled to end at the synchronization point in BB bb_5. The abstract view on our modeling of wait barriers is shown in Figure 4.6b. Both the case when we cannot determine the certain task and waiting for any task are not suitable for static scheduling. For the former case, we do not know the task whose execution should be enforced if feasible in order to decrease the waiting time. The latter case does

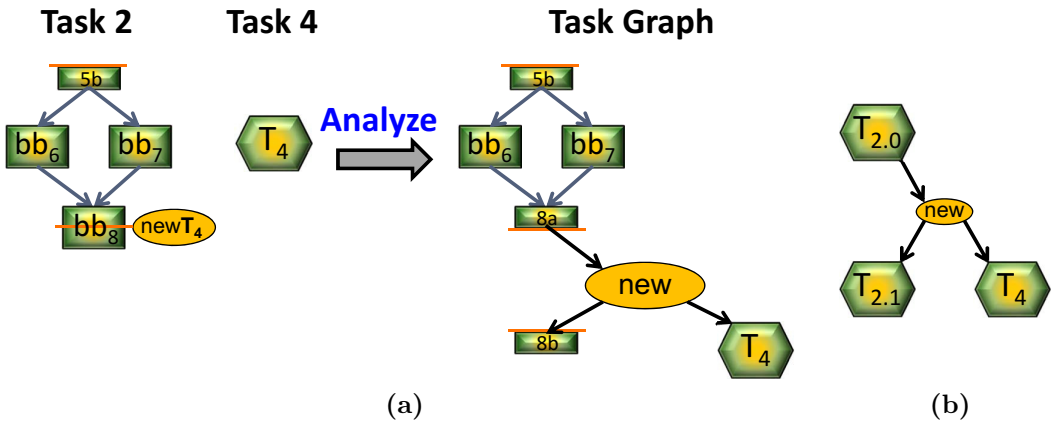


Figure 4.7: Analysis of task creation: **(a)** concrete and **(b)** abstract view

not impose a further constraint for scheduling because any other task should not be delayed (which is always the goal). Hence, we do not model these cases explicitly.

As important as the wait barrier, if not more than that, is the creation of tasks because it must not be overlooked during task allocation. It also can be seen as a barrier since a task cannot execute before its creation. Again, we take all creations into account. As discussed before, a task is created given the function to be executed by the task and a handle or the task ID is returned. If the creation is performed in a BB that post-dominates the entry BB or within an unconditional path in a loop where the loop header is a post-dominator of the entry BB, we create a new synchronization point at which the CFG of the corresponding function starts. Otherwise, we model the start of a task with a synchronization edge between the last synchronization point before the creation and the entry BB of the CFG of the task. Continuing the example in Figure 4.4, assume that the task T_2 creates a new task T_4 in BB bb_8. The result of our analysis is given in Figure 4.7a, showing the subtask of T_2 after the barrier in BB bb_5. We split the task T_2 at BB bb_8 and connect the parts and the CFG of task T_4 with synchronization edges. Additionally, we record the variable holding the handle or the task ID that is returned at creation to be able to determine task dependent behavior as introduced in Subsection 4.4.1. The abstract view on our modeling of task creation is shown in Figure 4.7b. If the creation of tasks occurs in conditional paths that will be taken depending on run-time values, we do not know whether a task is actually created. In this case, we tag the edge from the synchronization point to the CFG of the task with the information that it is a conditional creation.

Our static analysis of concurrent tasks that we have introduced in this subsection determines instructions that may be executed by each task as well as the communication and additional interdependencies between tasks. The result is a graph where the CFGs of tasks are connected via communication edges that are weighted with the volume to be transferred and are tagged with the information whether the communication is synchronous or asynchronous. To reflect further synchronization, we insert special synchronization points into

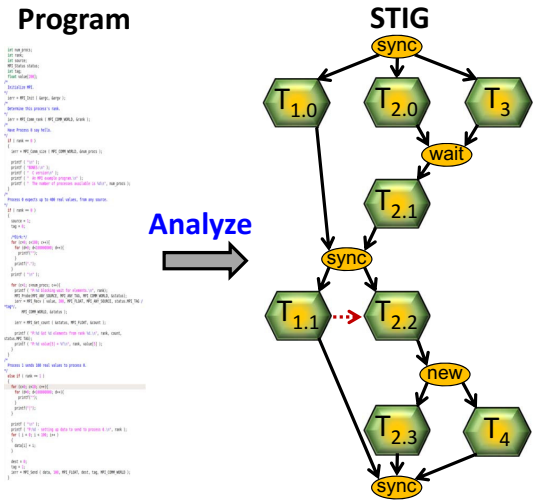


Figure 4.8: Synchronized Task Interaction Graph (STIG)

the graph and we split the participating tasks into subtasks. The result of our analysis is the STIG as given by Definition 4.7.

We model the start and the termination of an execution of applications with dummy synchronization points at begin and end of our STIG. Figure 4.8 depicts an exemplary STIG extracted by our analysis. It shows the combination of our discussed examples for modeling synchronization. First, task T_2 waits that task T_3 finishes its execution. Then, tasks T_1 and T_2 synchronize themselves and task T_2 afterwards creates a task T_4. Based on our STIG, we automatically build a cost model to rate the gain of alternative task mappings. In the following, we introduce our cost model.

4.4.2 Cost Model for Task Mapping

Our static analysis of concurrent tasks as discussed in the previous subsection yields a *Synchronized Task Interaction Graph* (*STIG*). Our STIG represents instructions that may be executed by tasks during execution phases of the program and models the communication and interdependencies between tasks. In particular, the STIG provides information about communication or synchronization partners, the points of execution where communication or synchronization arises, and the transferred volume of each communication point. From the *Machine Learning Phase* of our framework MaCAPA, we obtain predictor functions for unknown loop iteration counts, recursion frequencies, execution times, and the PE that executes a task most efficiently. On this basis, we automatically establish a cost model to predict computational costs of tasks on PEs and communication costs between tasks. This cost model is vital for our framework MaCAPA to rate the gain of alternative task mappings. We formally define our cost model as follows.

Definition 4.8 (Cost model)**.** *The cost model CM w. r. t. a program is a tuple*

$$CM := (\mathcal{T}_s, \texttt{comm}_{time}, \texttt{sync}, \texttt{time}, \texttt{bestPE})$$

where

- $\mathcal{T}_s$ *is the set of all subtasks as given by Definition 4.6,*
- *the relation* $\texttt{comm}_{time} \subseteq \mathcal{T}_s \times \mathcal{T}_s \times \mathbb{R}^{n^2} \times \{sync, async\}$ *denotes the communication overheads between subtasks for the possible allocations together with the information whether the communication is synchronous or asynchronous where n is the number of PEs of the target architecture,*
- *the relation* `sync` *as given by Definition 4.5 represents synchronization between subtasks,*
- *the function* $\texttt{time} : \mathcal{T}_s \to \mathbb{R}^d$ *returns the predicted execution times of a subtask on d different PEs of the target architecture ($d \in \mathbb{N} \wedge d \leq n$ where n is the number of PEs of the target architecture) via the machine learned functions $pred_{t_p}$ that we have defined in Subsection 4.3.4, and*
- *the function* $\texttt{bestPE} : \mathcal{T}_s \to \{p \in \mathbb{N} \,|\, p = 1, \ldots, d\}$ *returns for a subtask the predicted best performing PE of d different PEs of the target architecture via the function $pred_p$ as introduced in Subsection 4.3.4.*

As we have introduced in Subsection 4.3.4, we establish for each different PE of the target architecture a machine learned predictor function for execution times of tasks. To derive computational costs, we extract code features for each subtask between the synchronization points separately and we apply our predictor functions to each feature vector. From this, we obtain for each subtask a vector with predicted execution times, where each element represents the computational cost when the subtask is allocated to a particular PE. Additionally, we apply the predictor function for the best performing PE to each feature vector, which gives a prediction for the PEs that execute the subtasks most efficiently. If the best performing PE is not available when a subtask should be allocated there (because other tasks are yet allocated there), we examine the vector of predicted execution times for determining the next best PEs.

To derive the communication costs between tasks, we first determine the total communication amount. To that end, we analyze the surrounding CFG at communication points whether the communication occurs within loops or recursive functions. This enables us to derive the *frequency* of communication between tasks. If communication occurs within loops with statically determinable loop bounds, we use the iteration count as frequency of communication. If these loops do not have statically determinable loop bounds, we extract their features. Then, we apply the predictor function $pred_l$ for iteration counts that we have introduced in Subsection 4.3.1 to the feature vectors. Since the learning goal was classification, we choose the mean of the classified range of values as representative and use it as predictions for the communication frequency. If communication occurs within a recursive function, we extract its code features, apply the predictor function $pred_f$ for recursion frequencies as

introduced in Subsection 4.3.2 to the feature vector, and use the representative of the predicted class as communication frequency. If the communication is both within a loop and within a recursive function, we take the multiplication of both frequencies as prediction. In the case that the communication is neither in a loop nor in a recursive function, we use the static execution frequency computed according to the algorithm of Wu and Larus [WL94] as communication frequency. From this, we compute for each point of communication the *total communication amount* as the predicted frequency times the communication volume (provided by our STIG).

The communication overhead between two tasks depends not only on the communication amount but also on the rate to transfer the data. The transfer rate is relative to the bandwidth between two PEs, or between a PE and its main memory in case of communication through shared variables, respectively. Hence, the overhead also depends on the allocation of tasks to PEs. We therefore compute communication overheads for all possible allocations of tasks. That is, we calculate the overheads as the total amount divided by the bandwidth given that two tasks are allocated to particular PEs. If there is no communication medium between two different PEs, i. e., when the bandwidth is zero, we set the communication overhead to infinite. We thus obtain the predicted communication times for all combinations of allocations, which establishes the relation $\texttt{comm}_{time}$ in our cost model as given by Definition 4.8. The bandwidth between two PEs can be automatically obtained during profiling in the *Analysis Phase* of MaCAPA via the asymptotic throughput of communication with different amounts, averaged over a certain number of iterations. The bandwidth between a PE and its main memory is a feature for learning the best performing PE using the extended variant.

In summary, our cost model represents the cost of executing an application. As given by Definition 4.8, it holds the representations of the subtasks, the communication costs between subtasks, execution times of the subtasks between synchronization points, and the best performing PE for each subtask (see Figure 4.9). On the right of Figure 4.9, our cost model is shown. For each subtask i, t_i denotes its execution times on PEs given by the function `time` in our cost model, i. e., it is a vector where an element t_i^P represents the time on PE P. The PE that executes a subtask i most efficiently as given by the function `bestPE` in our cost model is depicted as pe_i. The annotation c_{ij} of a communication edge denotes possible communication overheads, i. e., it is a vector where an element $c_{ij}^{P_1,P_2}$ represents the communication time when the subtask i is allocated to PE P_1 and the subtask j is allocated to PE P_2. Based on our cost model, we perform communication- and power-aware task mapping, which we present in the following subsection.

4.4.3 Task Graph Mapping based on Cost Model

Within this thesis, we propose a general framework MaCAPA that aims at improving the mapping of concurrent applications to parallel architectures. To this end, we automatically establish a precise cost model based on machine

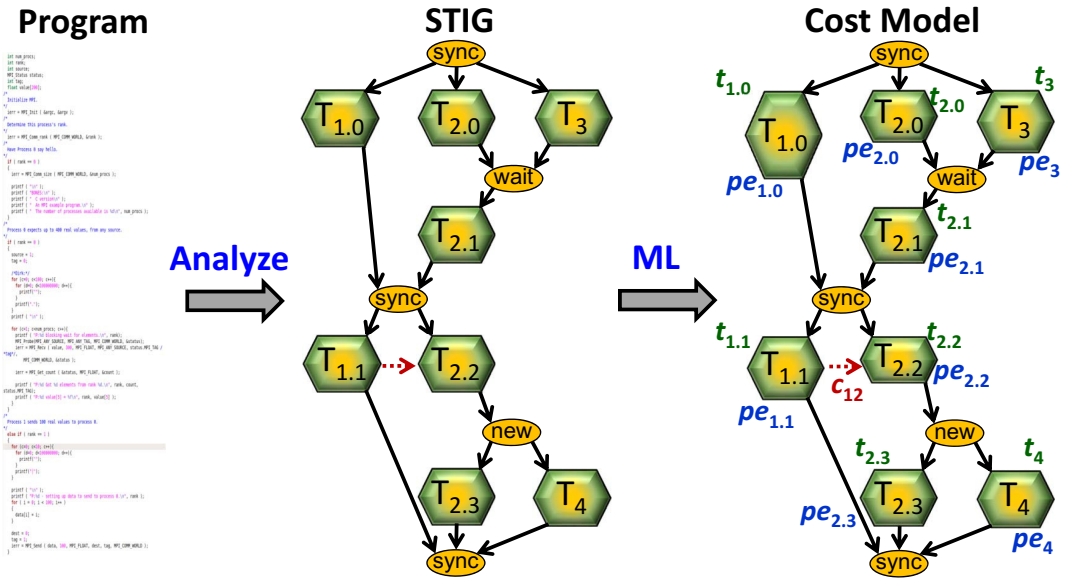


Figure 4.9: Cost model

learned predictor functions for expectable run-time behavior, which we have introduced in the previous subsection. The final step of the *Application Phase* of MaCAPA consists of mapping an application to the target architecture. This involves to schedule the parallelly executable tasks of the application and to allocate the tasks to the PEs of the architecture, which is crucial for the resulting run-time performance. We have developed a heuristic communication- and power-aware mapping algorithm that employs our ML based cost model. We automatically predict with the help of our cost model the resulting performance improvement of alternative task mappings and select the mapping with the highest gain.

Basically, our algorithm performs task mapping on the subtasks between synchronization points. It first assigns *priorities* to the subtasks that reflect their predicted computational costs as well as interdependencies and communication costs with other tasks. The subtasks are then scheduled based on their priority. For each subtask, our algorithm determines the PE that executes the subtask most efficiently while considering its communication costs. Our heuristics predicts the best PE that is available, thereby determining whether it is possible to map more than one task to the same PE without performance degradation. By doing this, the power consumption of the target architecture can be reduced because other PEs potentially can be put in a low-power state. Our heuristics also considers the cost for the reallocation at run-time to other PEs because reallocation can be advantageous in terms of computation and communication costs. For that purpose, our framework MaCAPA provides a run-time environment that performs the reallocation based on our static predictions if it is beneficial. Furthermore, our run-time environment allocates conditionally created tasks to the best performing available PE. Hence, our mapping algorithm not only improves an initial static mapping but also enables the improvement of the dynamic mapping of tasks.

Input to our mapping algorithm (see Algorithm 4.5) are the PEs of the target architecture and our cost model CM as given by Definition 4.8. In the following, we use the terms "task" and "subtask" interchangeably when the meaning is clear. Prior to the allocation of tasks, our algorithm determines execution time ranges of the phases of the program. That is, we determine for each synchronization point s (given by the relation `sync` in CM) the expected point in time when it will be reached during execution, which we call $exp(s)$, and whether this point in time is postponable without performance degradation, which we call $post(s)$. We use both information to decide whether it is advantageous to allocate more than one task to the same PE.

First, we order the synchronization points in our cost model CM by their dominance concerning the ID dependent CFGs of the tasks (Line 1). The obtained list *syncList* hence begins with the dummy synchronization point representing the start of an application and ends with the dummy synchronization point representing the termination of an application. We compute for each synchronization point *sync* in the list, i. e., "top-down", the expected point in time $exp(sync)$ when it will be reached during execution (Lines 3 and 4). The notation $i \rightarrow sync$ in Algorithm 4.5 denotes that a task i ends at the synchronization point *sync*. Likewise, the notation $s \rightarrow i$ denotes that a task i starts at the synchronization point s. The point in time is given by the maximal time after executing an incoming task on its best performing PE. The time after executing an incoming task is the sum of the cost for executing the task and the point in time of its preceding synchronization point (since at this point the task starts its execution). For the dummy synchronization point representing the start of an application, *exp* defaults to zero since there are no incoming tasks.

The `cost` of a subtask i on a PE P (Line 20) reflects the predicted execution time t_i^P on this PE together with the cost for reallocation when the subtask before is allocated to a different PE and the sum of its communication times. The execution time t_i^P is given by the result of the function `time` in CM at position of P and a communication time $c_{cj}^{P,P2}$ is given by the function $\texttt{comm}_{time}$ in CM. Since no allocation was performed during calculation of the expected points of time, the sum of communication times uses the maximal overhead, i. e., a scenario where the communication partner is allocated to a PE having the lowest bandwidth to the assumed PE of the subtask. Furthermore, the cost for reallocation is zero since the subtask is not allocated (Line 26).

We then compute for each synchronization point s to which amount $post(s)$ the expected point in time is postponable without performance degradation. The postponable criterion is calculated in reverse order of *syncList*, i. e., "bottom-up" (Lines 5 and 6). It is given by the minimal time at disposal after the synchronization point when an outgoing task is allocated to its worst performing PE. The time at disposal is the difference between the worst execution time of a task and the point in time where the task ends, relative to

Algorithm 4.5: Static task mapping

Input : PEs PE_i and cost model CM
Result : Allocation of scheduled tasks

1 $syncList$ = `orderSyncPointsByDominance(`CM`)`;
/* *compute expected and postponable point in time of synchronization* */
3 **foreach** synchronization point $sync$ in $syncList$ **do**
4 $\quad \exp(sync) = \max_{i \text{ with } i \to sync} \{\min_{PE} \texttt{cost(}i,\ PE\texttt{)} + \exp(s)\}$;
5 **foreach** synchronization point s in `reverse(`$syncList$`)` **do**
6 $\quad \text{post}(s) = \max\{0, \min_{\substack{i \text{ with} \\ s \to i}} \{ \exp(sync) - \max_{i \to sync} \max_{PE} \texttt{cost(}i,\ PE\texttt{)}\} - \exp(s)\}$;
7 $\quad$ **foreach** subtask i in CM with $s \to i$ **do** /* *assign priorities* */
8 $\quad\quad \text{prio}(i) = \bar{t}_i + \sum_j \bar{c_{ij}} + \max_{\substack{j \text{ with} \\ i \to sync \to j}} \text{prio}(j) + \max_{j \text{ with } c_{ij}} \bar{t}_j - \max_{\substack{j \text{ with } c_{ij} \\ \text{synchron}}} \bar{t}_j$;
9 **foreach** PE PE_i **do** $\text{avail}(PE_i) = 0$;
10 $taskList$ = `orderTasksByPrio(`CM`)`;
11 **foreach** subtask i in $taskList$ **do**
$\quad$ /* pe_i *executes task* i *most efficiently, try to allocate task* i *there* */
13 $\quad$ **if** $\max\{\text{avail}(pe_i), \exp(s)\} + \texttt{cost(}i, pe_i\texttt{)} \leq \exp(sync) + \text{post}(sync)$
$\quad$ **then** /* *best performing* pe_i *can be chosen* */
14 $\quad\quad bestPE = pe_i$;
15 $\quad$ **else** /* *take the best PE that is available* */
16 $\quad\quad bestPE = \arg\min_{PE}\{\max\{\text{avail}(PE), \exp(s)\} + \texttt{cost(}i,\ PE\texttt{)}\}$;
17 $\quad \text{alloc}(i) = bestPE$; /* *perform allocation* */
18 $\quad \text{avail}(bestPE) = \max\{\text{avail}(bestPE), \exp(s)\} + \texttt{cost(}i,\ bestPE\texttt{)}$;

19 **function** `cost(`subtask i, PE P`)`
20 $\quad \text{cost} = t_i^P + \texttt{costRealloc(}i,\ P\texttt{)} + \sum_{c_{ij}} \begin{cases} \max_{PE} c_{ij}^{P,PE} \text{ if } j \text{ not allocated} \\ c_{ij}^{P,P_2} \text{ if alloc}(j) == P_2 \end{cases}$;
21 $\quad$ **return** cost;

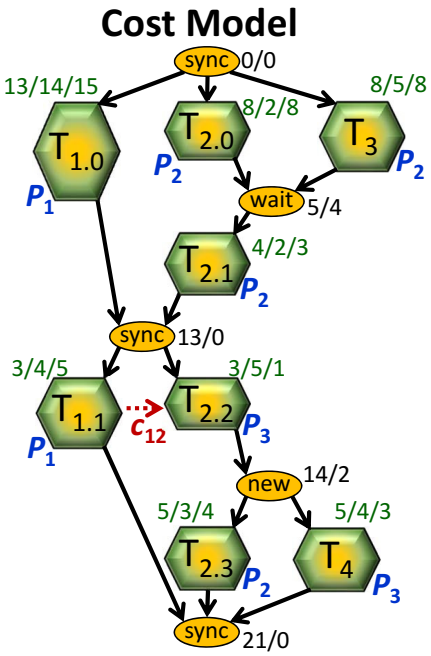
22 **function** `costRealloc(`subtask i, PE P`)`
23 $\quad$ **if** i is the first subtask **then** $\text{cost} = 0$;
24 $\quad$ **else**
25 $\quad\quad j$ = `getPartBefore(`i`)`;
26 $\quad\quad$ **if** j not allocated **then** $\text{cost} = 0$; /* *computing* exp *and* post */
27 $\quad\quad$ **else**
28 $\quad\quad\quad$ **if** $\text{alloc}(j) == P$ **then** $\text{cost} = 0$;
29 $\quad\quad\quad$ **else**
30 $\quad\quad\quad\quad \text{cost} = \frac{\texttt{size(}j,\ i\texttt{)}}{bandwidth(\text{alloc}(j), P)}$;
31 $\quad$ **return** cost;

the point in time where the task starts. Because the time at disposal can be negative, we bound it with zero as a minimum[12].

For the priority of a task i, which we also compute bottom-up (Line 8), we add up its average execution time on PEs $\bar{t_i}$ and the sum of its average communication times $\bar{c_{ij}}$. Both terms together reflect the execution cost of the task. The priority also adds up the maximal priority of tasks that cannot execute before task i has finished its execution. Thus, a higher priority is given to a task where a delay of its execution would also delay dependent tasks with high computational costs. It can be interpreted as the execution cost of the critical path in our STIG from task i until the end of the executions of all its descendants. Thereby, a decreasing order of priorities provides a topological order of the tasks. The last two terms that are included in the priority reflect the dependencies caused by communication. Because data must be computed before it can be communicated, we increase the priority of task i by the maximal average execution time of tasks that receive data. Thus, the execution of task i is favored over executions of tasks that wait for data since we schedule tasks based on their priority. If the communication is synchronous, i. e., the communication partners rendezvous at the communication point, the execution of task i should not be favored. Instead, the partners should execute simultaneously. We therefore decrease the priority by the maximal average execution time of tasks that synchronously receive data.

For the allocation of tasks to PEs, we record with $avail(P)$ at which time a PE P is available. We initialize this information with zero (Line 9) and we sort the tasks in decreasing order of their priorities (Line 10). We then allocate the tasks, beginning at the task with the highest priority (Lines 11 to 18).

In Figure 4.10, we show a cost model with exemplary values of execution times on PEs and the predicted best performing PE for each subtask, given that the target architecture has three PEs P_1, P_2, and P_3. The times are separated by slashes and are ordered according to the PEs (the scale unit of the time is irrelevant). The vector for the communication overhead between the tasks $T_{1.1}$ and $T_{2.2}$ is shown in the table below, where each column represents a possible allocation (the first two rows) with the associated communication time in the last row. The values next to synchronization points are the computed values of exp and



$T_{1.1}$	P_1	P_1	P_1	P_2	P_2	P_2	P_3	P_3	P_3
$T_{2.2}$	P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_3	P_3
c_{12}	1	5	3	5	1	4	3	4	1

Fig. 4.10: Exemplary cost model

[12]The time at disposal can be negative, e. g., when there is only one outgoing task that has a higher maximal cost than its minimal cost.

post (before and after the slash, respectively). The expected point in execution time of the wait barrier is five, which is caused by the execution time of task T_3 on its best performing PE P_2. The expected time when the tasks $T_{1.0}$ and $T_{2.1}$ synchronize is 13, caused by the task $T_{1.0}$. Therefore, the point in time of the wait barrier can be postponed. It is the difference between 13 and the worst execution time of task $T_{2.1}$ (i. e., four on PE P_1) minus the point in time, which results in four. Similarly, the expected point in time of the barrier where the task T_4 is created is 14 (13 plus one for the execution of $T_{2.2}$ on P_3). The best execution times of $T_{2.3}$ and T_4 are three. Since they start execution at time 14, both will finish earliest at time 17. The task $T_{1.1}$ has a minimal `cost` of eight on P_1 because on P_2, $t_{1.1}^{P_2} = 4$ with maximal communication time five and on P_3, $t_{1.1}^{P_3} = 5$ with maximal communication time four. The expected point of the dummy synchronization representing the termination is hence $13 + 8 = 21$. The creation of T_4 is thus postponable by two (calculated via $max\{0, min\{21-5, 21-5\} - 14\} = max\{0, 2\} = 2$).

The basic idea of our allocation algorithm (Lines 13 to 17 of Algorithm 4.5) is to decide whether a task i can be allocated to its best performing PE pe_i even if other tasks are yet allocated there. If this is not advantageous, we choose the best *available* PE. The best performing PE pe_i is determined with the function `bestPE` in our cost model CM as given by Definition 4.8. Considering a task i, the task can start execution at its preceding synchronization point s, i. e., at $exp(s)$. On a particular PE P, the execution can start when it is available, i. e., at $avail(P)$, but no earlier than $exp(s)$. That is, task i can start its execution on PE P at $\max\{avail(P), exp(s)\}$. The execution on the best performing PE pe_i has the cost `cost(`*i, pe*$_i$`)` and the point in time of the synchronization *sync* where it should be finished is $exp(sync)$. This point can be postponed without performance degradation by $post(sync)$. Thus, the condition in Line 13 evaluates whether it is advantageous to choose the best performing PE pe_i for task i, no matter if another task is yet allocated there. If not, we determine in Line 16 the PE at which the task i causes the minimal run-time overhead while executing when it is available. Then, we allocate the task to the best available PE that we have determined (Line 17) and we update the time when the PE is available (Line 18).

During allocation, the `cost` of execution we consider also includes the cost for reallocation when the subtask before was allocated to another PE, which we compute via `costRealloc` (Lines 22 to 31). The reallocation cost is zero if it is the first part (Line 23) or if both subtasks will be allocated to the same PE (Line 28). Otherwise, we derive the cost via the `size` to be transferred to the new PE and the bandwidth between the PEs (Line 30). This size depends on the amount of data computed by the subtasks before that must be made available on the new PE and on the execution state of the subtasks before that must be maintained. The execution state includes the stack containing the functions in the calling sequences and their current execution state. Since this run-time information is not statically determinable in general, we use predictions for the size of the execution state. During execution, local variables and parameters for each function in the calling sequence are stored on the stack. To predict their size, we examine the static CG of the whole task. We

add up the static sizes of parameters and local variables of the functions that are in the static calling sequence of the CG up to the current subtask. If a function in the sequence is recursive, we multiply the static sizes with our ML prediction for the recursion frequency. The data that must be made available consists of all global variables and the dynamically allocated variables in the heap space of all subtasks before. For the former, we add up the sum of their static sizes and for the latter, whose size also is not statically determinable, we use the sum of the total communication amounts that the subtasks before send and receive. The rationale for this is the requirement that communicated data must be stored anywhere. Finally, the remaining instructions to be executed must be transferred. We derive this size from the CFG of the task and also add it up. Our prediction for the reallocation cost is then given by the total size divided by the bandwidth.

Continuing our example in Figure 4.10, the priorities that we compute for the tasks are $prio(T_{2.3}) = prio(T_4) = 4$, $prio(T_{1.1}) = 10$, $prio(T_{2.2}) = 7$, $prio(T_{2.1}) = 13$, $prio(T_{1.0}) = 24$, $prio(T_{2.0}) = 19$, and $prio(T_3) = 20$. The order in which our mapping algorithm allocates the tasks hence are $T_{1.0}$, T_3, $T_{2.0}$, $T_{2.1}$, $T_{1.1}$, $T_{2.2}$, $T_{2.3}$, and T_4[13]. We allocate the task $T_{1.0}$ to its best performing PE P_1 since its `cost` on P_1 is equal to the time of the synchronization point at which its execution ends and so the condition in Line 13 evaluates to true. We then allocate the task T_3 to its best PE P_2 and set $avail(P_2)$ to 5. For allocating the task $T_{2.0}$ to its best PE P_2, the condition in Line 13 evaluates to true because the wait barrier is postponable without performance degradation. Therefore, $T_{2.0}$ is allocated to P_2 (although another task is yet allocated there) and $avail(P_2)$ is set to 7. This example demonstrates that our developed algorithm is power-aware because doing so enables to put PE P_3 in a low power state at this time of execution. Similarly, we also allocate the subtasks $T_{2.1}$ and $T_{1.1}$ to their best PEs, i. e., to the same PEs where their parts before has been allocated. For $T_{2.2}$, PE P_3 is predicted to perform best (which was not the best PE for the part before). Assume a reallocation cost of two. Then, this subtask will be reallocated because the time after execution is not greater than the postponed time of the wait barrier and $avail(P_3)$ is set to 16. Now assume that the cost for the reallocation of $T_{2.3}$ is five (because, e. g., the subtask before has received data from $T_{1.1}$). Then, $T_{2.3}$ will not be allocated to its best performing PE because the total `cost` of eight when starting execution at time 14 exceeds the point in time when it should finish. Instead, the minimal cost is caused on P_3, i. e., on the same PE as the subtask before. Consequently, $T_{2.3}$ is allocated on P_3 and $avail(P_3)$ is set to $16 + 4 = 20$. The last task to be allocated is T_4, for which P_3 is predicted to be the best. Because P_3 is not available under the constraint to finish at time 21, the task T_4 will be allocated to PE P_2.

At run-time of applications, our framework MaCAPA provides an environment that coordinates the allocation. It is crucial for the resulting run-time performance of the application that the additional overhead of our run-time environment is as low as possible. When the algorithm as introduced above

[13] Tie-breaking regarding the priorities is done randomly, so it would be possible to allocate T_4 before $T_{2.3}$.

determines a new allocation at a synchronization point, we automatically insert an instruction in the IR that signals our run-time environment to perform the allocation. If no reallocation at a synchronization point is determined, such as at the wait barrier of the exemplary cost model in Figure 4.10, no signal is send. To process the allocation, we store the mapping decisions and our run-time environment simply has to utilize this information, which can be evaluated very fast. Likewise, when tasks are created conditionally (i. e., the synchronization edge is marked conditionally), our run-time environment is signaled. For these conditionally created tasks, we provide a fast dynamic allocation algorithm. We allocate a conditionally created task i to its best performing PE pe_i if no other task is yet allocated there. We know how many tasks are allocated to PEs because our run-time environment performs the allocation and records this information. If other tasks are yet allocated on this best PE, we choose the best predicted PE that is free. If at least one task is allocated on each PE, we allocate the task on the PE with the minimal run-time cost under the current workload of the PEs. That is, the task i will be allocated to a PE P with

$$P = \arg\min_{PE} \texttt{cost}(i,\ PE) \cdot \texttt{numAllocated}(PE) \ ,$$

where `numAllocated` returns the number of currently allocated tasks on a PE.

In conclusion, our dynamic allocation algorithm complements the static algorithm to improve the mapping for all created tasks at run-time of applications. Our developed mapping algorithms thus are *complete* because they always find a solution for the mapping problem such that each task is mapped to a PE that is available at run-time[14]. Since a decreasing order of our calculated priorities provides a topological order of the tasks concerning their dependencies and since we statically map the tasks based on this order, all dependencies between tasks are met. Hence, our static mapping algorithm *ensures program correctness*. The dynamic mapping algorithm does not affect the order of tasks because a task is mapped as soon as it is created. Hence, the dynamic algorithm also does not violate dependencies between tasks and thus also preserves the semantics of the program.

Both the static and the dynamic algorithm provide fast heuristics for deriving the mapping. The static allocation as well as the computation of *exp* and *post* of synchronization points have a worst-case complexity of $\mathcal{O}(n \cdot t \cdot p^2)$, where n is the number of subtasks, t is the number of tasks, and p is the number of PEs of the target architecture. This holds because the function `cost`, which has a worst-case complexity of $\mathcal{O}(t \cdot p)$ for computing the sum of the maximal communication time (on p PEs) over t possible communication partners[15], is called for each of the n subtasks p times (on each PE). The computation of priorities also has a worst-case complexity of $\mathcal{O}(n \cdot t \cdot p^2)$ because for n subtasks, the sum over t possible communication partners of the

[14] A solution always exists because we do not consider real-time systems with deadlines that can be missed. Since all tasks are mapped, our algorithms find a solution.

[15] At most t (and not n) possible communication partners exist because communication cannot cross synchronization points.

average communication time concerning p^2 possible allocations must be calculated. The dynamic algorithm has a worst-case complexity of $\mathcal{O}(t \cdot p^2)$, where t is the number of tasks and p is the number of PEs of the target architecture since the function `cost` is called for p PEs.

In this section, we have presented the *Application Phase* of MaCAPA. We have described how our framework MaCAPA automatically analyzes the expectable run-time behavior of programs, which includes executed instructions by each task as well as the possible communication and further interdependencies between tasks. Based on this and our ML predictors for required run-time behavior that were established during the *Training Phase* of MaCAPA, a precise cost model is built. This cost model is used to rate the gain of alternative allocations. As a result, our general framework MaCAPA enables a power- and communication-aware mapping of concurrent applications to parallel architectures. In the next section, we first summarize the benefits of our conceptual framework. Then, we revisit the objectives of our framework and decide whether the corresponding criteria are fulfilled by MaCAPA.

4.5 Summary

In this chapter, we have presented our general framework for *Machine Learning based Mapping of Concurrent Applications to Parallel Architectures* (*MaCAPA*) that comprises two basic phases, a *Training Phase* and an *Application Phase*. The *Training Phase*, which has to be performed one time only per architecture, is composed of an *Analysis Phase* and a *Machine Learning Phase*. In the *Analysis Phase*, we analyze code features and collect run-time behavior. Both information constitute the input for our ML algorithms during the *Machine Learning Phase* to establish predictors for required run-time behavior. During the *Application Phase* of MaCAPA, a precise cost model is derived based on the established predictors, which then is employed to map tasks power- and communication-aware to the PEs. This has the advantage to focus the mapping on expectable run-time behavior since the learned predictors capture run-time behavior from several profiling runs. Furthermore, our approach mitigates the drawbacks of both profiling, which extends every compilation, and manual annotations, which are tedious obligations for programmers, because learning is done automatically in a one-off *Training Phase* per architecture decoupled from the compilation. Thus, the framework does not require user intervention and does not increase the compile time by training. Our approach to automatically generate architecture- and application-specific heuristics for power- and communication-aware task mapping also scales for large applications and is fully retargetable, i. e., the learning algorithms depend solely on static features and can be applied without modification to any architecture.

In Section 1.2, we defined nine objectives to assess the quality of our general framework MaCAPA. In the following, we revisit each objective in turn and discuss whether MaCAPA fulfills the corresponding criteria.

- **Automatic Application of the Analyses** All analyses of our framework MaCAPA do not require manual annotations of values given by a user. The analyses are hence fully automatically applicable without user intervention.

- **Precision of the Analyses** The analyses during the *Analysis Phase* of MaCAPA regarding the extraction of static code features and profiling of run-time behavior yield exact results. Our static analysis of concurrent tasks yields as precise results as statically possible because we separate data dependent behavior from behavior that depends on the ID of a task. The analyses of the run-time overhead are based on ML predictors that relate static features to dynamic behavior. The ML predictors were built with observations from several profiling runs, which capture expectable behavior. Hence, our ML based analyses are more precise than pure static analyses because the latter must consider all possible cases how a program can behave and therefore may drastically over-approximate.

- **Efficient Compilation** Our analyses of MaCAPA regarding the extraction of static code features have a linear complexity in the number of instructions because they solely have to count occurrences of constructs in the IR. The analysis of concurrent tasks of applications including our *Least Fixed-point Algorithm* to determine the behavior of a task does not impose a larger compile time overhead than usual program analyses [NNH99, Kno08] as used within compiler environments for compilation of complex real-world applications[16]. Hence, analytical results are obtained in an acceptable timeframe. Since the *Training Phase* of MaCAPA is decoupled from the compilation, the compile time is not increased by training. In conclusion, our framework ensures an efficient compilation.

- **Continuous Compilation Flow** The framework MaCAPA preserves a continuous compilation flow because all analyses and applied transformations have been integrated into the compiler framework.

- **Scalability of the Heuristics** Our developed task mapping heuristics relies on ML predictors for required run-time behavior. The scalability of ML methods has been shown with the application of machine learning to large databases, which is called *data mining*. Our ML predictors are based on classification tree learning, which is one of the most popular methods in data mining [Ste09]. Our heuristics provide a static and a dynamic mapping algorithm that also scale for large task graphs because the algorithms do not have to explore the complete search space of all possible allocations. Instead, we provide fast greedy algorithms. As we

[16] The complexity of a *Least Fixed-point Algorithm* depends on the size of the property space that must be explored. In our case, the algorithm iterates when it is first determined that an argument of a function call holds an environmental variable. That is, the iteration count is bounded by the sum of the number of arguments over all function calls. In each iteration, a CFG is traversed. Hence, our algorithm has a worst-case complexity of $\mathcal{O}(c \cdot s)$, where c is the maximal size of the CFGs of functions that are defined by the application and s is the sum of the number of arguments.

have discussed in Subsection 4.4.3, the static mapping algorithm has a worst-case complexity of $\mathcal{O}(n \cdot t \cdot p^2)$ and the dynamic mapping algorithm has a worst-case complexity of $\mathcal{O}(t \cdot p^2)$, where n is the number of subtasks, t is the number of tasks, and p is the number of PEs of the target architecture.

- **Generality of Parallel Programming Models** We solely assume that the program to be mapped is written in a parallel programming language, e. g., UPC, which realizes the *Distributed Shared Memory Model*, MPI, which realizes the *Message Passing Model*, and threaded C or OpenMP, both realizing the *Shared Memory Model*. The framework hence is general enough to be applicable to various parallel programming models.

- **Generality of Target Architectures** Our framework MaCAPA extends conventional compilers whose back ends generate machine code for existing architectures. For each new architecture, our approach requires nothing but a one-off *Training Phase* of MaCAPA. Because we do not otherwise constrain the target architecture, the framework targets architectures with unrestricted structure.

- **Generality of the Cost Model** The established cost model to rate the gain of alternative mappings uses hardware-dependent information about the execution times on PEs and about the communication overhead. Both is obtained during the one-time-only *Training Phase* of MaCAPA per architecture. Hence, our cost model rates the expected gain of the mapping to different architectures.

- **Generality of Application Domains** For our experiments that show the accuracy of our approach, we have used a considerable number of programs from various benchmark suites which encompass different real-world application domains. Furthermore, we do not impose theoretical restrictions on the application domains. This shows the general applicability of our framework MaCAPA.

Thus, the general framework MaCAPA presented in this chapter fulfills all our objectives to provide automatic, efficient, and highly scalable methods for statically analyzing the expected run-time behavior of applications and, based on this, for optimizing the run-time performance of applications. Our framework can be instantiated for different parallel programming models in combination with different target architectures. For example, it can be applied to OpenMP programs that realize the shared memory programming model in combination with any architecture having a shared memory, such as multi-core processors or *symmetric multiprocessors* (*SMPs*). The framework can also be instantiated to map MPI programs to arbitrary architectures. MPI realizes the message passing programming model used for large systems with distributed memory. Because shared memory implementations of MPI are possible [MPI12], it may also be used for shared memory architectures. Hence, our framework MaCAPA also enables us to improve the mapping of MPI programs to, e. g., *Multi-Processor Systems-on-Chip* (*MPSoCs*). Furthermore, the

framework can be used for improving the mapping of UPC programs to architectures with a logically shared address space. MaCAPA can be instantiated for hybrid parallel programming models as well, e. g., for a combination of message passing and shared memory access (which may be realized using MPI together with threads or with OpenMP). In the next chapter, we show how our general framework is applied to improve the mapping of MPI programs to processor networks.

5 Intelligent Mapping of MPI Programs to Processor Networks

The allocation of processes to *Processing Elements* (*PEs*) is a vital part of mapping concurrent applications to parallel architectures. Improving this mapping requires information about execution times of processes during run-time of applications, which is not known in advance at compile time. Furthermore, the mapping must take into account interprocessor communication at run-time, whose overheads have emerged as the major performance limitation in parallel applications. In the previous chapter, we have shown our general framework MaCAPA for *Machine Learning based **Ma**pping of **C**oncurrent **A**pplications to **P**arallel **A**rchitectures*. Our approach for learning the run-time behavior of tasks automatically generates predictors, which relate static code features to the dynamic run-time behavior, thus providing the compiler with intelligence about required run-time information. By using these predictors as heuristics to rate the gain of alternative mappings, our framework enables the automatic improvement of the run-time performance of concurrent applications. In the following, we present an instantiation of MaCAPA to improve the mapping of *Message Passing Interface* (*MPI*) programs to arbitrary processor networks.

With this thesis, we aim at, among other things, reducing the communication overhead between parallelly executable tasks at run-time. When applying our general framework to the mapping of MPI processes to processor networks, this plays a major role for improving the resulting performance. Considering processor networks, the communication medium between the processors, such as Ethernet or Myrinet, has a much lower bandwidth than the bandwidth between a PE and the main memory. The impact of communication latency on the whole program performance is hence evident. The communication overhead can be decreased if MPI processes that communicate most frequently with each other are allocated to PEs with a high bandwidth connection. Our machine learned information for the mapping of MPI processes to a processor network improves the initial allocation because processes that communicate most frequently can be automatically allocated to PEs with the minimal communication latency in between. Furthermore, it enables the reallocation of processes at run-time whenever their computation or communication behavior changes. Both features are huge improvements to current MPI implemen-

tations because even the best available MPI implementations cannot ensure that, for instance, frequently communicating processes are placed close to each other [Trä02]. For heterogeneous processor networks, we additionally provide predictions about the PE that executes a process most efficiently. As a result, our approach for automatically allocating these MPI processes to appropriate PEs can yield a notable performance gain. We have published the basic idea of using *Machine Learning* (*ML*) to improve the mapping of MPI processes in [TG12a].

5.1 Overview

In this chapter, we define how we use machine learned run-time behavior predictions to allocate MPI processes to PEs in a heterogeneous processor network. For the instantiation of MaCAPA to the MPI mapping problem, the *Training Phase* of MaCAPA remains unchanged. The *Training Phase* comprises the *Analysis Phase*, which we have presented in Section 4.2, and the *Machine Learning Phase*, which we have presented in Section 4.3. During the *Analysis Phase*, which is the first step of the *Training Phase*, we extract static features and we obtain considered run-time behavior via profiling. Both are input to the second step of the *Training Phase*, i. e., to the *Machine Learning Phase* where ML models are constructed that relate static features to dynamic behavior. During the *Application Phase* of the instantiated framework MaCAPA, we use these models as precise and fast heuristics for power- and communication-aware mappings of MPI processes to PEs of a processor network. For that purpose, we provide both a static algorithm, which performs the initial allocation, and a dynamic algorithm, which allocates conditionally created processes and reallocates processes based on our static predictions if it is beneficial. In this chapter, we mainly focus on the *Application Phase* since the instantiated *Training Phase* is identical to the *Training Phase* of the general framework.

Figure 5.1 illustrates the steps of the instantiated MaCAPA framework for mapping MPI processes to a processor network. Input to our framework is the source code of an MPI application which we analyze. Based on the analysis of synchronization between the processes, we split each participating process into two parts per synchronization point, the part of the process before the synchronization and the part of the process after the synchronization. From this, we build an *MPI process graph*, which is an instantiated *Synchronized Task Interaction Graph* (*STIG*) of our general framework, as shown in Figure 5.1 on the left (we introduced the STIG in Definition 4.7 on Page 107). The hexagons represent parts of processes and edges represent dependencies. The parts are labeled with the first number being the ID of the corresponding MPI process and the second number representing the part of the process. To reflect synchronization, we insert synchronization points into the graph. In Figure 5.1, three processes start to execute the application. First, the processes P_2 and P_3 synchronize with each other and then P_1 and P_2. Afterwards,

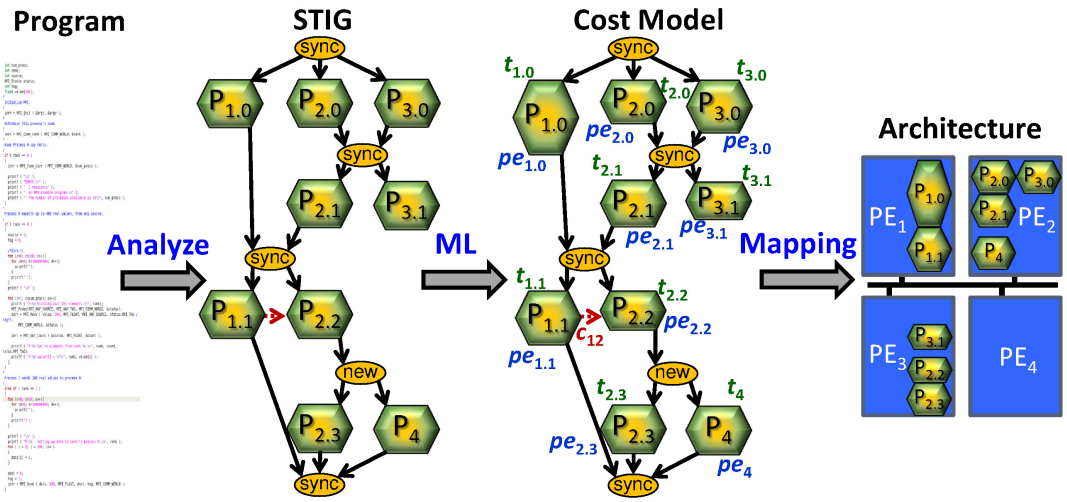


Figure 5.1: Mapping of MPI processes to parallel architectures

the latter processes communicate with each other (indicated by a red dotted arrow). Finally, a process P_4 is created by process P_2.

With the help of our established ML predictor functions from the *Training Phase* of MaCAPA, we derive close approximations for expectable run-time overhead from the analyzed behavior of processes. To build a cost model w. r. t. the application to be mapped, we annotate the MPI process graph with our ML based predictions. In Figure 5.1, we show our cost model. For each part of a process P_x, t_x denotes its execution times on different PEs and pe_x denotes the PE that executes the part most efficiently. The communication overhead w. r. t. different possible allocations between two processes x and y (which is a matrix of communication times) is depicted as c_{xy} near the red dotted communication arrow.

The final step of our instantiated framework MaCAPA consists of mapping the MPI application to the given heterogeneous target architecture, which is guided by our cost model. The essential idea of our mapping heuristics is to decide whether a process can be allocated to its best performing PE even if other processes are already allocated there. The decision is based on the cost of the process, which reflects its computational cost, its communication overhead, and the cost for reallocation at run-time. If the mapping to its best PE is not advantageous, we choose the best *available* PE. On the right of Figure 5.1, we show an exemplary network of PEs with the black line between the PEs denoting the communication medium (the longer the line the longer the communication latency). Assume the execution time of the part of the process $P_{1.0}$ in Figure 5.1 is predicted to be longer than the sum of execution times of $P_{2.0}$, $P_{3.0}$, and $P_{2.1}$ and the latter parts have the same predicted best PE PE_2. Our mapping heuristics is able to recognize this and therefore decides to allocate these parts to the same PE instead of performing workload balancing. This enables to put PE_3 in a low power state while the parts $P_{2.0}$ and $P_{3.0}$ are executing. This example demonstrates that our mapping algorithm is power-aware in the sense that it keeps PEs empty if this

does not lead to a performance degradation. Our heuristics also considers the reallocation of processes at run-time. If, for instance, the reallocation cost for the part $P_{2.2}$ (which communicates with $P_{1.1}$) is lower than the performance gain from a shorter communication latency, the process P_2 will be reallocated from PE_2 to PE_3.

In the following, we first briefly introduce the MPI standard. Then, we describe the *Application Phase* of our instantiated framework MaCAPA. For the *Application Phase*, we start with our static analysis of MPI programs in Section 5.3. Afterwards, we describe how our ML based cost model is used to determine the mapping in Section 5.4.

5.2 Message Passing Interface Standard

The MPI standard [MPI12] is an interface specification of a message-passing library. An MPI program consists of autonomous processes, which are identified with a unique ID, called *rank* in the MPI terminology. Each process is an instance of the same MPI program. The processes execute the code in a *Multiple Instruction Multiple Data* (*MIMD*) style, i. e., the instructions actually executed by each process do not need to be identical. Which portion of a program will be executed by a process can be selected via the rank.

In the MPI standard, the ID of a process is related to a *group* in a *communicator*. `MPI_COMM_WORLD` is the initially defined universe communicator with a global group for all processes. A group is used within a communicator to describe the participants in a communication universe and to rank such participants. The MPI run-time environment assigns consecutive integer values to the processes in a group, starting at zero, to enable a unique identification. To get the unique ID within a group, i. e., the rank, a process must invoke the MPI library call `MPI_Comm_rank` where the communicator is an argument of this call. The result of this call is assigned to a variable and can be compared at guards for conditional paths in the CFG whether its value is in a certain range. Groups can be created dynamically based on ranks of existing groups where the ranking related to the new group is not necessarily identical to the ranking in the existing group. As a consequence, an MPI process can be identified by more than one unique ID.

Message passing is commonly categorized according to the communication partners as *one-to-one*, *one-to-all*, *all-to-one*, or *all-to-all*. The first category is called a *point-to-point* communication in the MPI terminology and the latter ones are termed as *collective*. An operation is collective if all processes in a group need to invoke the operation. Since groups can be dynamically created at run-time, the collective operations establish in fact a *one-to-many*, *many-to-one*, or *many-to-many* communication.

The *point-to-point* communication is realized with different calls for the sending and the receiving process. The processes communicate via calls to MPI communication primitives like `MPI_Send` and `MPI_Recv`. A receive operation

may accept messages from an *arbitrary* sender. In contrast, a send operation for point-to-point communication must specify a *unique* receiver. These calls take as arguments among others the communicator, the type and the number of elements of transferred data. The MPI standard furthermore offers to *tag* message calls of point-to-point communication with an integer number that connects a send operation with a receive operation.

The MPI standard classifies *collective* operations into five types:

broadcast A one-to-many communication, where the whole content of the send buffer is sent from one member to all members of a group.

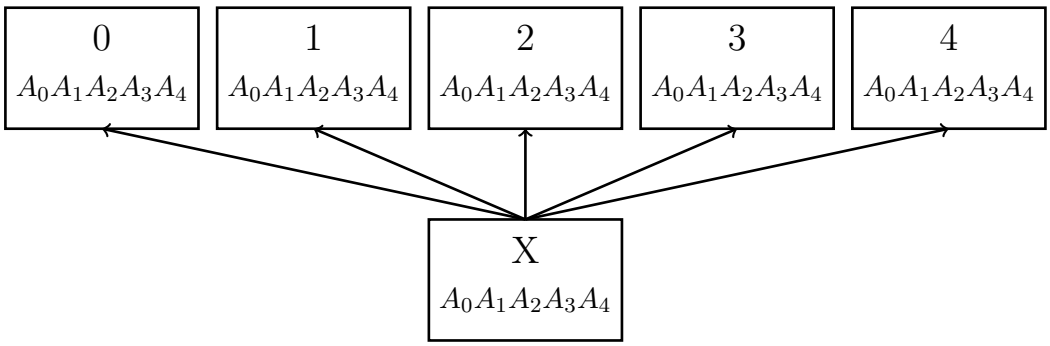
scatter A one-to-many communication, where data is scattered from one member to all members of a group, i. e., the i-th segment of the send buffer is sent to the i-th process.

gather A many-to-one communication, where data is gathered from all members of a group and sent to one member.

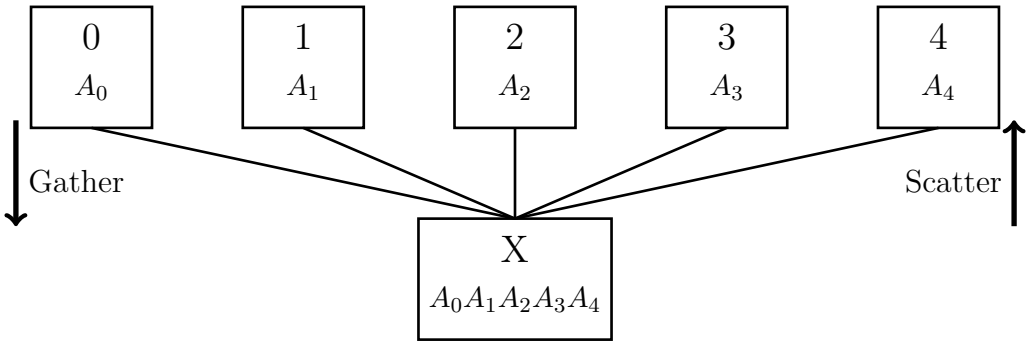
allgather A variation of gather, where all members of a group receive the result, i. e., a many-to-many communication.

complete exchange A many-to-many communication, where data is scattered and gathered from all members to all members of a group.

In Figure 5.2a, we show the principle of a one-to-many *broadcast* communication. Each rectangle represents a process having the rank as depicted in the



(a) Broadcast communication in MPI

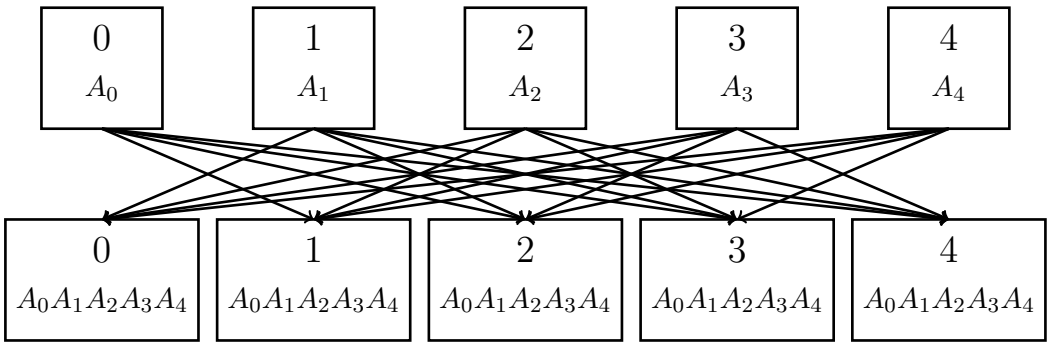


(b) Gather data from all to one / scatter data from one to all members

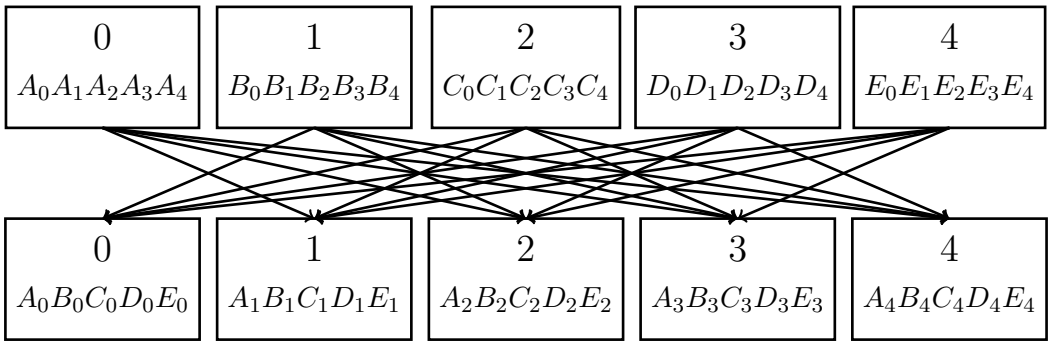
Figure 5.2: One-to-many / many-to-one communication in MPI

upper row within the rectangle. The lower rows within the rectangles constitute elements in the communication buffers and the arrows represent data exchange. As can be seen, a process X sends the whole content of its buffer to the five members of the group. In contrast to that, a *scatter* operation, which is depicted in Figure 5.2b, sends only the i-th segment of the buffer to the i-th process. The collective communication of type *gather*, which is also shown in Figure 5.2b, is the inverse operation to scatter. Additionally to the number of sent elements, both the scatter and the gather communication calls have a parameter to specify the number of elements that are received. Similarly, the types of the many-to-many communication, which are shown in Figure 5.3, differ in the number of sent and received elements. They therefore also take the number of received elements as an argument to the corresponding call. In Figure 5.3a, the type *allgather* of the many-to-many communication is shown. It is a variant of gather where the whole content of each send buffer is transferred to all members of a group. The *complete exchange* of the buffers is exemplified in Figure 5.3b. It is an extension of *allgather* to the case where each process sends distinct data to each of the receivers. The j-th segment of the send buffer from process i is received by process j as the i-th segment in its receive buffer.

An example for an MPI program that performs collective communication within dynamically created groups is given in Figure 5.4. After starting MPI with a call to `MPI_init` at Line 10, the processes obtain their rank in the universe communicator of all processes and assign their values to the respec-



(a) Gather data from all members to all members of a group



(b) Complete exchange from all members to all members of a group

Figure 5.3: Many-to-many communication in MPI

```
1  int main(int argc, char *argv)
2  {
3    int rank, newRank;
4    int firstGroup[4] = {0, 1, 2, 3}, secondGroup[4] = {4, 5, 6, 7};
5    int sendBuf, recvBuf;
6    MPI_Group globalGroup, newGroup;
7    MPI_Comm newComm;

9    /* starts MPI */
10   MPI_Init(&argc, &argv);

12   /* get current process id */
13   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

15   /* get global group of all processes */
16   MPI_Comm_group(MPI_COMM_WORLD, &globalGroup);

18   /* create two distinct groups based upon ranks */
19   if(rank < 4) MPI_Group_incl(globalGroup, 4, firstGroup, &newGroup);
20   else MPI_Group_incl(globalGroup, 4, secondGroup, &newGroup);

22   /* create new communicator newComm and get newRank for newComm */
23   MPI_Comm_create(MPI_COMM_WORLD, newGroup, &newComm);
24   MPI_Comm_rank(newComm, &newRank);

26   /* perform collective communications at newComm */
27   sendBuf = rank;
28   MPI_Allreduce(&sendBuf, &recvBuf, 1, MPI_INT, MPI_SUM, newComm);

30   printf("rank= %d newRank= %d recvBuf= %d\n", rank, newRank, recvBuf);

32   /* terminates MPI */
33   MPI_Finalize();
34 }
```

Figure 5.4: Example for the creation of groups and communication in MPI

tive rank variables (Line 13). At Line 16, the global group of all processes is assigned to the variable *globalGroup*. Based on the ranks in this group, two distinct groups are created with calls to `MPI_Group_incl` at Lines 19 and 20. This library call takes as first argument the old group of participants from which the new group is created and as second argument the number of participants that shall be in the new group. The third argument is an array of values that denote the processes (by their rank) to appear in the new group. The arrays that are passed to the function call are initialized at Line 4 such that the processes with ranks zero to three will be in the first group and the processes with ranks four to seven in the second group. As specified in the MPI standard, the index in the array determines the new rank of the process in the

new group. As a result, the processes with ranks four to seven in the global group will have the new ranks zero to three in the new group (which they assign to the variable *newRank* at Line 24). At Line 28, all processes invoke a call to `MPI_Allreduce` within the new communicators in the groups (given as last argument). This call performs a reduction operation over all send buffers and distributes the result to all receive buffers. Hence, it is a variation of the *allgather* collective communication. The reduction operation to be performed, which is passed as penultimate argument, is the sum over the send buffers (to which the ranks of the processes are assigned at Line 27). In this example, the members of the first group just have the value 6 in their receive buffers (the sum of the ranks 0 to 3) and the members of the second group the value 22 (the sum of the ranks 4 to 7).

To start an MPI program, the user has to specify how many processes shall execute the program. The MPI run-time environment then starts this number of processes and is responsible for the communication data transport and the coordination between the processes. Furthermore, the user must define the available PEs of the target architecture since otherwise, all processes are started at the host where the user invoked the program start. A PE to which an MPI process can be allocated is a node in the processor network such as a core of a processor with multiple CPUs or a processor on a *Multi-Processor System-on-Chip* (*MPSoC*). The user can also specify to which PE a process will be allocated, in which case the given allocation is fixed even if the run-time behavior indicates that reallocation to another PE would be beneficial. If no advice for the allocation is given, the MPI run-time daemon starts the processes in a Round Robin fashion on available PEs.

In the following, we present our approach to automatically improve the allocation of MPI processes to the PEs of a processor network.

5.3 Analysis of MPI Programs

In the application of our framework MaCAPA to the MPI mapping problem, the analysis of concurrent tasks has to be realized. This involves to analyze the instructions that may be executed by an MPI process as well as the communication and interdependencies between MPI processes. As a result, we obtain an *MPI process graph*, which is an instantiation of our *Synchronized Task Interaction Graph* (*STIG*) to MPI processes. In the following, we describe our analyses that establish the MPI process graph.

5.3.1 Executed Instructions by MPI Processes

A major challenge of mapping MPI programs to a processor network is to precisely analyze the possible behavior of the processes. Based on this information, we are able to derive a cost model for the computational costs of processes and the communication costs between processes. As a prerequisite,

we have to separate run-time behavior that is specific for a process, which we call the *ID dependent behavior*, from data dependent run-time behavior. In this subsection, we present our analysis of executed instructions by MPI processes, which yields the ID dependent behavior.

Since the MPI processes execute in *Single Program Multiple Data* (*SPMD*) mode, they are given by the whole program. That is, they are realized by the `main` function of the program (whereby the unique rank of each process defines which part of the program is executed by which process). Hence, we have a single CFG for all processes, which we analyze. Furthermore, we solely have to extract one single *Call Graph* (*CG*) – the CG cg_{main} of the `main` function – instead of a distinct CG for each process. Based on cg_{main}, we determine the ID dependent behavior of each process via the analysis of conditional paths in the CFG cfg_{main} of the `main` function. That is, we apply Algorithm 4.4 to cfg_{main} and cg_{main}.

We provided a function `DUchain` for our Algorithm 4.4. The function analyzes the variables that are defined by the MPI run-time environment (such as the number of executing processes or the IDs of processes), which we call *environmental variables*. For the instantiation of MaCAPA to the MPI mapping problem, the function `DUchain` additionally considers that processes can be identified via more than one ID. The MPI processes must invoke the MPI library call `MPI_Comm_rank` to get their unique ID, i. e., their rank. Our function `DUchain` therefore updates the set `env` of environmental variables whenever a call to `MPI_Comm_rank` occurs. If the communicator given as argument to this call is the global communicator for all processes, our function `DUchain` includes the corresponding variable in the set `env` of environmental variables. If it is not the global communicator, it represents a communication universe of a dynamically created group. In this case, our function `DUchain` analyzes the ranking within the newly created group. As described above (in Subsection 5.2), the ranking is given by the indices in the array of group members. For a static initialization of the array (as shown in Figure 5.4 at Line 4), the analysis is trivial. For initializations that are performed within a loop, we provide a loop induction variable analysis to determine the values of elements at each iteration. If we can derive the exact content of the array from the *Intermediate Representation* (*IR*), the variable with the new rank is included in the set `env` and our function `DUchain` provides a mapping from the original ranks to newly defined ranks. Since it is not always possible to statically analyze the exact content of an array, our algorithm safely over-approximates the ID dependent behavior in this case as follows. If different initializations occur in conditional paths in the CFG for the same array and we can determine the values of the elements in the array for all initializations, we take the union of the values as set of potential group members. The mapping of an original rank yields in this case a set of potential new ranks.

We defined (in Subsection 4.4.1) a predicate `pred` on conditional paths in the CFG to derive whether a path remains in the ID dependent CFG. The predicate also handles that processes can be identified via more than one ID and that our function `DUchain` may over-approximate the mapping

from original ranks to newly defined ranks. In this case, conditional paths will remain in the ID dependent CFG of the process if our predicate `pred` evaluates to true for at least one element in the set of potential ranks. If newly defined ranks can neither be statically determined nor approximated as described above, the variable with the new rank is not included in the set `env` of environmental variables. As a consequence, conditional paths in the CFG that depend on this variable are considered to be data dependent (instead of being specific for a process). Our predicate `pred` thus evaluates to true and the corresponding path will remain in the ID dependent CFG of the process (even if the actual value of the rank in the new group does not fulfill the condition).

Initially, the set of environmental variables is empty. If no function call to `MPI_Comm_rank` occurs in the whole program, the set remains empty. In this case, our Algorithm 4.4 yields the same CFG and CG for all processes, namely CFG $cfg_t^{ID} = cfg_{main}$ and CG $cg_t^{ID} = cg_{main}$. This is the correct result because all processes will execute the same instructions.

In this subsection, we have shown how we determine the ID dependent behavior of MPI processes as closely as possible. After having analyzed both the CFG and the CG, we use the ID dependent behavior to analyze the communication between the MPI processes, which we discuss in the following subsection.

5.3.2 Communication between MPI Processes

To enable the minimization of communication overhead at run-time, we have to analyze the communication behavior between MPI processes. We have shown (in Subsection 4.4.1) that a precise communication analysis requires to identify the communication partners and the amount of communication. In the instantiation of MaCAPA to the mapping of MPI processes to processor networks, the communication is explicit because the processes cannot use shared variables to communicate. That is, the processes use message passing, which we analyze. In the following, we present our communication analysis.

MPI uses communicators and groups to define which collection of processes may communicate with each other. With the capability for creating groups at run-time, the need for an analysis of dynamically created groups arises for a precise analysis of communication partners. We have introduced the analysis of dynamically created groups in the previous subsection. If we can derive a mapping from original ranks to the ranks in the new group, we precisely know the (possible) partners. Otherwise, we must initially assume that all processes are in the new group and hence may communicate with each other. Due to the fact that MPI requires collective communication calls to be performed by *all* members of the group, we are able to narrow down the members of the new group. When a collective call for a new group is not in the ID dependent CFG of a process, we definitely know that this process cannot be in the group. In this case, we remove the rank from the group such that the process is not included in the set of communication partners at other collective calls for this group. We can furthermore narrow down the members of a group with our

analysis of interdependencies that divides the execution into *phases* (which are delimited by *synchronization barriers*). Since communication cannot cross a barrier, we also remove a process from a group when a collective call is not in the correct phase.

The analysis of the partners for one-to-many and many-to-many communication is then straightforward. For the former category, the sender of the message is the process whose CFG we are currently visiting and for the latter, all members of the group send a message. We then create communication edges between the corresponding partners in the current group.

Our analysis of many-to-one and one-to-one communication additionally determines the single receiver of the message, which must be given to the message passing call. The partner for MPI communication is specified with its rank but it does not need to be a constant in the IR of the application. Our analysis of the communication partner either statically identifies the exact partner or a set of possible partners if the exact partner cannot be statically determined. In the latter case, we again narrow down the possible partners to the set of processes that are in the current group and that perform the collective operation in the current phase. Additionally, the specification of the MPI standard allows us to limit the set of possible communication partners even more accurately. The MPI standard not only requires that *all* members of a group must invoke the same collective operation but also that the arguments of this operation are compatible between all invocations. In addition to the receiver (which of course must be the same), the type of the transferred data, which is also an argument of the message passing call, must be equal across all invocations. Consequently, we remove a process from a group if it does not perform the call with the correct type.

For a point-to-point communication, our analysis of the single receiver additionally considers the *tag* with which send and receive calls can be connected. For a number of send calls within a communicator, there must exist matching receive calls for the same data type such that the number of sent elements is equal to the number of received elements[1]. Since the MPI standard specifies that a receiver may accept messages from an arbitrary sender, the rank of the sender does not need to be given to the receive call as argument (technically, it can be `MPI_ANY_SOURCE`). The tag can then be used to identify the sender. This tag allows us to further limit the set of possible partners if we cannot statically analyze the exact partner. If we can statically derive that the values of a tag of matching message calls between possible partners are equal, we have determined the exact partners. In this case, we create a corresponding communication edge between both partners. Otherwise, we create communication edges between all possible partners.

To realize the communication analysis of our general framework MaCAPA, we also analyze the communication *volume* and we weight each edge with its

[1]The number of elements given to the send and the matching receive call need not be equal because, e. g., sending 100 elements in one message can be answered by receiving 1 element in a loop that iterates 100 times. We thus cannot use the number of elements transferred to match a send call with a receive call.

respective volume. All variants of message exchange in the MPI standard, i. e., the point-to-point and the collective communication, require the number of transferred elements and their data type to be given as argument to the call. The communication volume is then the size of the data type times the number of transferred elements of that type. The number of elements transferred depends on the semantics of the call. As with the ID of the communication partner, the value of the number of elements does not have to be a constant in the IR. Similarly, the size of the data type need not be constant. Though the size of a basic data type is a compiler known constant, the MPI standard allows the creation of data types at run-time for which it is not always possible to determine their size. The collective communication calls except for the broadcast communication take the number of sent and received elements as arguments and, additionally to the data type of sent elements, the data type of received elements. As a consequence, we have two possibilities to analyze the communication volume for this kind of communication. The volume to be transferred is either the number of sent elements times their data type size or the number of received elements times their data type size, divided by the number of communication partners. We analyze both the former for send calls and the latter for matching receive calls to determine the communication volume. In the case where we cannot determine the number of transferred elements or their size for a communication call[2], we assume a fixed amount as volume. As reasonable value for this fixed amount is the average of analyzed volumes of the current application or, if the volumes of all communications calls cannot be determined, the actual transferred volumes during profiling in the *Analysis Phase* of MaCAPA, averaged over all programs.

The result of our communication analysis so far is a graph where the message exchange between MPI processes is modeled with communication edges that are weighted with the communication volume. It is the instantiated *task communication graph* of our general framework MaCAPA as given by Definition 4.4 with tasks realized as MPI processes. The communication can be seen as an interdependency between the processes because the data must be sent before another process can access this data. Additionally to this enforced ordering, the processes may have further interdependencies that affect the runtime performance of a program. In the following, we present our analysis of these interdependencies.

5.3.3 Interdependencies between MPI Processes

With our analysis of communication partners and volumes as shown in the previous subsection, we are able to identify communication costs between MPI processes. This allows us to reduce the communication overheads at run-

[2]Note that we analyze the volume per communication call and not the total number of elements to be transferred during communication, e. g., via looping over array elements or via recursion over recursive structures. This is accomplished with our corresponding ML predictor functions for loop iteration counts and recursion frequencies. Hence, we do not have to consider, e. g., a loop bound analysis to determine the number of transferred elements.

time of MPI programs, which plays a major role on improving the resulting performance. In this subsection, we present our analysis of interdependencies between MPI processes. This analysis enables us to determine precedence constraints on the execution of processes that also may influence the runtime performance of programs. Precedence constraints obviously arise from message exchange – the sending process must compute the data before it can send the data and the receiving process must wait for message arrival before the data can be used. Besides this data dependency, MPI processes may additionally synchronize with each other, either caused by particular message calls or independently from communication via explicit barriers.

Considering the exchange of messages between processes, the MPI standard specifies different *modes* for point-to-point communication. The *standard mode* for message exchange are *blocking* operations. That is, the operations do not return until the message data is stored either into the matching receive buffer or into a temporary system buffer. Message buffering decouples the send and receive operations. A blocking send can complete as soon as the message has been buffered, even if no matching receive has been executed by the receiver. The MPI standard also specifies a *nonblocking mode*. A procedure is nonblocking if it may return before the operation completes[3]. Both the blocking and nonblocking modes do not impose precedence constraints other than the ordering enforced by data dependencies. There are also *synchronous* communication operations, for instance `MPI_Ssend` and `MPI_Srecv`. The synchronous send will complete only if the corresponding receive operation has started to receive the message. If both sends and receives are blocking operations then the use of the synchronous mode provides a synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. If MPI processes communicate in synchronous mode, we tag the corresponding communication edges with this information.

Other synchronization calls exist independently from communication where a number of processes perform a rendezvous. To capture the execution before and after synchronization points, we split each of the participating processes into multiple parts and we create synchronization edges between all parts. That is, our analysis of synchronization between MPI processes establishes the relation `sync` as given by Definition 4.5 for our general framework MaCAPA and each part of an MPI process is an instance of a *subtask* as given by Definition 4.6. We have distinguished three types of a rendezvous:

1. A *synchronization barrier*, where the participating tasks will block their computation on encountering the barrier during execution until the last participating task encounters the barrier.

2. A *wait barrier*, where a task must wait until other tasks have finished their execution.

3. A *new barrier*, where a task creates other tasks.

[3] An operation completes when the user is allowed to reuse resources and any output buffers have been updated.

In the instantiation of MaCAPA to the mapping of MPI programs to processor networks, we do not have to consider *wait barriers* because the MPI standard does not specify a corresponding operation. Each process is an instance of the whole program and completes normal execution at the end of the `main` function. A process also can abort the execution by itself via calls to, e. g., an `exit` function or `MPI_Abort`[4] before reaching the end of the `main` function. However, this represents error handling and terminates the whole MPI program.

Synchronization barriers can be used to ensure that a number of processes reach a certain point of execution. This enforces an ordering constraint on the execution of processes. As we have discussed for our general framework MaCAPA, we consider solely synchronization barriers if they are in a BB that post-dominates the entry BB of the ID dependent CFG. This allows us to derive a unique dominance order of the barriers, which represents execution phases of the program. All synchronization barriers provided by the MPI standard, such as `MPI_Barrier` or `MPI_Win_fence`, are collective operations related to a group of processes. We first determine the participants of these barriers with our analysis of groups as presented above (in Subsection 5.3.1). Then, we split the processes into parts at the corresponding BBs of their CFGs and relates the parts together with the relation `sync`.

We also analyze the creation of processes and model it with *new barriers*. The creation of processes is specified by the MPI standard with two directives, namely `MPI_Comm_spawn` and `MPI_Comm_spawn_multiple`, which both are collective operations over a given communicator. Since MPI processes are instances of a whole program, the code to be executed by the created processes is given by (a name of) a precompiled binary together with arguments that are passed to this binary. Both directives additionally take as argument the number of processes to create. `MPI_Comm_spawn_multiple` is an extension to `MPI_Comm_spawn` where multiple executables can be specified to start, each by a different number of processes. We analyze whether the creation depends on run-time values. If this is the case, the creation is marked as being conditional. In the instantiation of MaCAPA to the MPI mapping problem, this encloses the case where the number of processes to create depends on run-time values.

All of the spawned processes have a same new communication universe `MPI_COMM_WORLD`, which is different from the given communicator passed to the directives. Since groups are related to communicators, the spawned processes constitute a group that is different to the *local* process group of the given communicator. Both directives yield a new communicator, called an *inter-communicator*, that connects the local group with the group of newly created processes. As a result, all processes in the local group are able to communicate with the newly created processes. The ordering of processes (which defines their ranks) in the group of this inter-communicator is the same as the ordering of the local group and of `MPI_COMM_WORLD` of the spawned processes. That is,

[4] `MPI_Abort` takes a communicator as argument, but it is not possible for an MPI implementation to abort only the processes represented by the communicator. The communicator argument is provided to allow for future extensions of MPI to environments with, for example, dynamic process management [MPI12].

all processes retain their ID regarding their own communication universe as well as regarding the inter-communicator. Message exchange initiated over an inter-communicator then targets a process of the other group (and not a process in the local communication universe). We use this information for our analysis of communication partners and volumes.

Our static analysis of MPI programs that we have introduced in this section yields an instantiation of the STIG as given by Definition 4.7. We call this instantiation an *MPI process graph*. It represents instructions that may be executed by MPI processes during execution phases of the program and models the communication and interdependencies between processes. Next, we derive a cost model from the process graph with our ML based predictions for execution times, communication overheads, and the PE that executes a process most efficiently. This cost model can then be used for communication-aware and power-efficient allocation of processes to PEs, which we present in the following section.

5.4 Cost Model based Process Graph Mapping

In this section, we show how we use our instantiated framework to automatically determine the mapping of MPI processes to PEs in a heterogeneous processor network. In the *Training Phase* of our general framework MaCAPA, which remains unchanged for the instantiation of MaCAPA to the MPI mapping problem, we employ ML techniques to automatically establish statistical models that relate static code features to considered run-time behavior. The basic idea of our framework is to derive predictions about required run-time information from these models, which then are used to assess the computational cost of processes and the communication cost between processes regarding alternative allocations. In particular, we obtain predictor functions for unknown loop iteration counts, recursion frequencies, execution times on different PEs, and the PE that executes an MPI process most efficiently. In the following, we describe how we construct a cost model by employing these predictor functions and, based on this, how we automatically derive the allocation of MPI processes to PEs.

First, we establish a cost model w. r. t. the current MPI program to be mapped in a given architectural setting. As specified in Definition 4.8, the cost model CM of our general framework is a tuple

$$CM := (\mathcal{T}_s, \texttt{comm}_{time}, \texttt{sync}, \texttt{time}, \texttt{bestPE}) \ .$$

It comprises the representations of the subtasks $\mathcal{T}_s$, the communication costs $\texttt{comm}_{time}$ and synchronization `sync` between subtasks, execution times of the subtasks between synchronization points (given by the function `time`), and the best performing PE for each subtask (given by the function `bestPE`). To establish an MPI cost model where tasks are realized by MPI processes, these elements of CM have to be instantiated as follows.

- Our static analysis of MPI programs, which we have introduced in the previous section, extracts an MPI process graph that models the possible interactions between MPI processes. The process graph provides information about the communication and synchronization between processes such as the partners, the points of execution where it arises, and the transferred volume of each communication point. From this process graph, we directly obtain the relation `sync` for our cost model, which represents synchronization between MPI processes, and the set $\mathcal{T}_s$ of all parts of processes between synchronization points.
- As defined in Subsection 4.3.4, we build a machine learned predictor function for the PE that executes the part of an MPI process most efficiently and for each different PE of the target architecture a predictor function for execution times. We apply these functions to (the features of) each part of the MPI processes, which gives predictions for its best performing PE and for its computational costs on available PEs. This yields the function `bestPE` and the function `time` of our cost model.
- As we have introduced in Subsection 4.4.1, we derive from the communication volume the *total amount* with the help of our predictor functions for loop iteration counts and recursion frequencies if communication occurs within loops or recursive functions. During profiling in the *Analysis Phase* of MaCAPA, we obtain the bandwidths between the PEs of the target architecture. From this and the derived total communication amount, we determine the *communication overhead* regarding the possible allocations of processes to the PEs. This yields the relation $\texttt{comm}_{time}$ of our cost model.

In summary, we are able to automatically derive the complete cost model conforming to Definition 4.8. We use this cost model for a communication-aware and power-efficient allocation of processes to PEs, which we discuss in the following.

With our instantiated framework MaCAPA, we automatically predict which allocation will result in the highest performance gain for the given application. To this end, we have developed a heuristic communication- and power-aware mapping algorithm that employs our ML based cost model, which we have defined in Algorithm 4.5. The inputs to our mapping algorithm are the available PEs of the target architecture[5] and our MPI cost model w. r. t. a given program. Additionally, the number of processes is provided by the user. The algorithm then proceeds as defined for our general framework MaCAPA. It basically predicts for each part of the processes the available PE that executes the part most efficiently, thereby determining whether it is possible to map more than one process to the same PE without performance degradation. To determine the mapping, our heuristics considers the cost of processes on PEs between synchronization points. This cost reflects their predicted execution times on particular PEs, communication overheads w. r. t. alternative allocations, and the cost for reallocation to other PEs at run-time. Potential points

[5]If a user does not define available PEs, we assume that the host where the program start was invoked is the only node in the processor network.

of execution where reallocation could be beneficial are the synchronization points given by our cost model. For each synchronization point, our algorithm derives the expected point in time when it will be reached during execution and whether this point in time is postponable without performance degradation. This is used to decide whether it is advantageous to allocate more than one process to the same PE.

Our instantiated framework MaCAPA extends the MPI run-time environment to enable the reallocation at run-time and the allocation of conditionally created tasks to their best performing available PEs. If our mapping heuristics determines a new allocation at synchronization points, the mapping decisions are stored and this is signaled to our run-time environment during execution of the program. At run-time, the allocation decision can be evaluated and processed with a low additional overhead, which is crucial for the resulting performance of the application.

For the dynamic allocation, i. e., if the creation of processes depends on run-time values, our instantiated framework MaCAPA provides the MPI run-time environment with a fast heuristics. Our allocation heuristics applies the three following mapping rules.

1. A process i will be allocated to its best performing PE pe_i if no other process is already allocated there.

2. If other processes are already allocated on this best PE, the process will be allocated to the predicted best PE that is free.

3. If at least one process is allocated on each PE, our heuristics chooses an allocation which causes the minimal run-time overhead. That is, the process will be allocated to the PE P with

$$P = \arg\min_{PE} \texttt{cost}(i,\ PE) \cdot \texttt{numAllocated}(PE)\ ,$$

   where `numAllocated` returns the number of currently allocated processes on a PE and `cost(`i, PE`)` returns the cost to execute process i on the PE.

Each of the three rules employs ML based predictions. The first rule uses the predictor function for the best performing PE and the second rule uses the predictor function for execution times. The third rule determines the `cost` with the predictor functions for loop iteration counts and recursion frequencies (to derive the communication overhead), and the predictor function for execution times. The dynamic creation of MPI processes has the peculiarity that the processes have to execute a precompiled binary for which no source code is given (except for the trivial case in which the binary is the same application as the one that we currently map). As a consequence, the CFGs of the processes from which we derive the feature vectors solely consist of a call that starts the binary. Nevertheless, we are able to sort the different PEs according to their performance with our ML models for the best performing PE and the execution times. To determine the communication overhead of dynamically created processes, we analyze the communication over the inter-communicator that connects the created processes with the group of processes that initiated the

creation. Since message exchange over an inter-communicator always targets a process of the other group and since collective operations have to be invoked by all group members, we can derive the communication between processes of the current application and created processes. That is, if a message reception or a collective operation over the inter-communicator occurs, we know that created processes have sent the data. We therefore use the analyzed total communication amount of these operations to calculate the communication overhead for the corresponding dynamically created processes. From this and the predictor function for execution times, we calculate the `cost` for executing a process as defined in Algorithm 4.5. Then, we choose with the third rule the PE that causes the minimal run-time overhead under the current workload (if rules one or two do not apply).

In this section, we have presented the final step of the *Application Phase* of our instantiated framework MaCAPA where MPI processes are automatically mapped to a given heterogeneous processor network. Based on the extracted MPI process graph, which is an instantiation of the STIG of our general framework MaCAPA, we first derive a precise cost model. The cost model employs predictor functions from the *Machine Learning Phase* to rate the gain of alternative mappings. This model is then used for a power- and communication-aware allocation of MPI processes to the PEs of the processor network.

5.5 Summary

In this chapter, we have described the instantiation of the general MaCAPA framework to the MPI mapping problem. The fundamental idea to automatically improve this mapping is to derive predictions about unknown run-time information via ML techniques that relate static code features to dynamic run-time behavior. The general framework MaCAPA enables the automatic improvement of mappings for different parallel programming models in combination with different target architectures. For the instantiation of MaCAPA to improve the mapping of MPI programs to heterogeneous processor networks, the *Training Phase* remains unchanged. The *Training Phase* of MaCAPA is required to establish ML based predictor functions for unknown run-time behavior. The *Application Phase*, which we have presented in this chapter, starts with the analysis of instructions that may be executed by MPI processes, which is a prerequisite for predicting the possible behavior of processes. On this basis, we identify the possible communication and interdependencies between processes. As interdependencies, we consider the synchronization between MPI processes and the dynamic creation of processes at run-time. With the help of our ML predictor functions from the *Training Phase* of MaCAPA, we derive close approximations for expectable run-time overhead from the analyzed behavior of processes. This is shown in Section 7.2, where we present an empirical evaluation of the precision of our predictor functions. Based on this, we derive a precise cost model. This cost model enables our mapping heuristics to select a solution of the mapping problem, which is expected to

yield the highest optimization gain. Our heuristics also considers the reallocation of processes at run-time. For conditionally created processes during run-time of the application, we provide a dynamic mapping heuristics that automatically selects the best performing available PE. Hence, our mapping algorithm not only improves an initial static mapping, but also enables the improvement of the dynamic mapping of MPI processes. As a result, the run-time performance of MPI programs can be significantly increased, which we show in Section 7.3. In the next chapter, we present the implementation of our instantiated framework MaCAPA.

6 Implementation

In this chapter, we describe the implementation of our instantiated MaCAPA framework. With this instantiation, we target the mapping of *Message Passing Interface* (*MPI*) programs to heterogeneous networks of processors. As a result, we obtain a complete experimental platform, which compiles an MPI application and automatically maps the processes that have to execute the resulting binary to the *Processing Elements* (*PEs*) of the processor network. The platform supports the full C language and the complete MPI 3.0 standard [MPI12]. It can cope with huge software in the area of *High Performance Computing* (*HPC*).

In the following, we first give an overview of the implemented platform and of the tools we used. Then, we show in more detail how the two basic phases of our instantiated framework are realized as implementation modules. We start with the *Training Phase* of MaCAPA, which comprises two steps. We describe its *Analysis Phase* in Subsection 6.2 and its *Machine Learning Phase* in Subsection 6.3. Then, we present the implementation of the *Application Phase* of MaCAPA in Subsection 6.4.

6.1 Overview

In this thesis, we have proposed a general framework MaCAPA for *Machine Learning based **Ma**pping of **C**oncurrent **A**pplications to **P**arallel **A**rchitectures*. In the previous chapter, we have presented its instantiation for improving the mapping of MPI programs to arbitrary processor networks. We have implemented the instantiated framework to obtain an executable platform with which we assess the actual performance improvement that can be achieved with our approach. Figure 6.1 depicts the structure of our implementation. The diagram corresponds to the overview of the general MaCAPA framework (compare Figure 4.2 on Page 71) and illustrates how the components of MaCAPA relate to implementation modules. The figure shows the main phases of the instantiated MaCAPA framework.

The *Analysis Phase* of MaCAPA is completely implemented within the *Compiler Development System* (*CoSy*). In the *Analysis Phase*, static code fea-

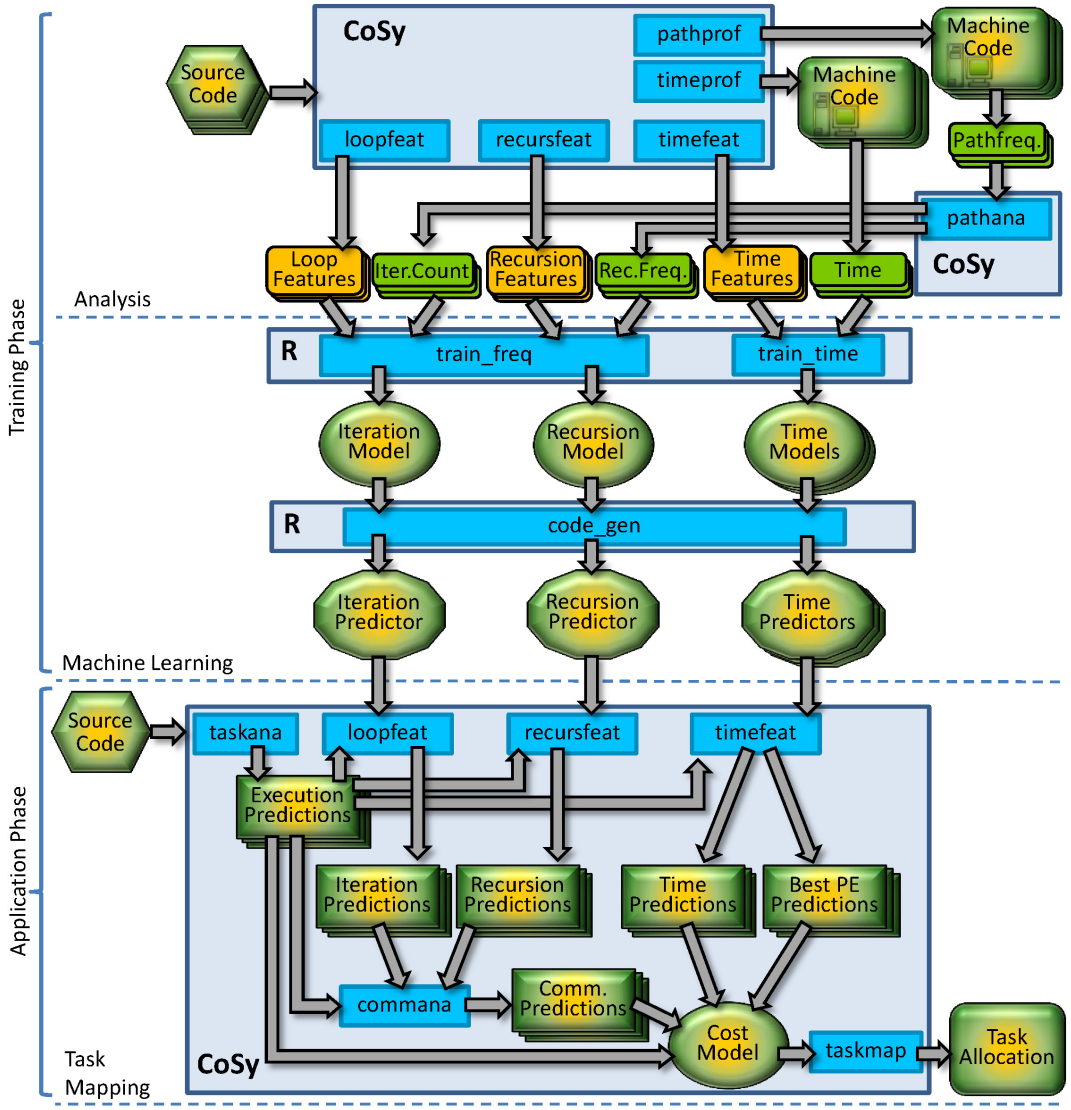


Figure 6.1: Implementation of *MaCAPA*

tures are collected, namely features for learning loop iteration counts (*loopfeat*), recursion frequencies (*recursfeat*), and execution times (*timefeat*). Additionally, corresponding run-time information is collected via profiling from a comprehensive suite of programs. For this purpose, the programs are instrumented to analyze and store observed values of regarded behavior. The implementation module *pathprof* instruments the *Intermediate Representation* (*IR*) to obtain execution frequencies of paths, from which we then derive loop iteration counts and recursion frequencies with the module *pathana*. The module *timeprof* instruments the IR to measure execution times.

The *Machine Learning Phase* is entirely implemented within the *R Project* environment for statistical computing [R12]. The training data (i. e., the pairs consisting of a feature vector and the corresponding correct outcome) are input to our *Machine Learning* (*ML*) techniques *train_freq* and *train_time*, which construct statistical models during the *Machine Learning Phase* of MaCAPA. The established models relate static code features to considered run-time be-

havior. In particular, we obtain a model for the loop iteration count, a model for the recursion frequency, and for each different PE of the processor network a model for execution times. From these models, our implementation module *code_gen* automatically generates executable predictors, which are used in the *Application Phase* of MaCAPA as fast heuristics.

The *Application Phase* of MaCAPA is entirely implemented within the *CoSy* compiler framework. During the *Application Phase*, the source code of an MPI application is first transformed by the compiler front end to an *Intermediate Representation* (*IR*). Our implementation module *taskana* then analyzes the parts of the program that may be executed by each MPI process. From this, *loopfeat* generates predictions for loop iteration counts and *recursfeat* generates predictions for recursion frequencies. Both are used by *commana* to predict communication overheads. Our implementation module *timefeat* generates predictions for the execution time and for the best performing PE. These predictions together with the predicted executed instructions and communication overheads establish our cost model. Finally, the allocation of MPI processes to PEs of the processor network is derived from the cost model by *taskmap*.

In the following, we first briefly introduce the external tools we used. Then, we consider each of the main phases of our instantiated framework MaCAPA in turn and describe notable aspects of our implementation.

External Tools

CoSy We use the modular, industrial strength *CoSy*© compiler development system by *ACE* [ACE13] as platform and extend it with our instantiated framework MaCAPA. The steps during compilation like, e. g., analyses and optimizations, are realized by reusable and reentrant modules called *engines*. The front end of *CoSy* first translates C source code to an IR, called the *Common CoSy Medium-level Intermediate Representation* (*CCMIR*). The compiler framework *CoSy* provides a rich functionality to define custom engines. The engines are written in C and can employ predefined macros for data flow and control flow analyses and for transformations of the CCMIR. After high-level optimizations (independent from the target architecture) are performed, a rule based code generator in the back end of *CoSy* lowers the CCMIR to a *Low-Level IR* (*LIR*). Then, target dependent analyses and optimizations of the LIR are performed aiming at improving the representation. In the last step within the back end of *CoSy*, assembler code is emitted from the LIR, which finally is translated to the executable binary (by the assembler and linker of the *GNU gcc*).

R Project The *R Project* environment for statistical computing [R12] provides a collection of statistical functions, which is made available as *packages* from the *Comprehensive R Archive Network*. We have drawn all regarded algorithms for classification and regression during the *Machine Learning Phase* from provided packages and have implemented them in the language *R*. For

classification learning, we use the *randomForest* package [LW02]. The different regression techniques we used for learning the execution times are provided by the packages *rminer* [Cor13] and *MASS* [VR02]. The language R offers a wide variety of facilities for data manipulation, calculation and graphical display. Technically, R is an expression language with object oriented features and properties of functional languages.

6.2 Analysis Phase

In the one-time-only *Analysis Phase* of MaCAPA, we collect code features by means of static analysis as well as dynamic run-time behavior through several profiling runs from a comprehensive suite of programs. As a result, we obtain tables of observations where one row represents one observation and the columns correspond to the features and, as a special column, the observed behavior. Each table constitutes the training data to build an ML model during the *Machine Learning Phase* of MaCAPA. In the following, we regard each step of acquisition of training data in turn.

6.2.1 Static Code Features

We collect features that count occurrences of IR constructs (we discussed the features in Subsection 4.2.1). Due to conditional branches in the CFG, one cannot expect that a construct will always be executed at run-time of the application. We therefore weight several features with the static execution frequency prediction of the corresponding IR construct. We compute the predictions based on the static branch prediction algorithm proposed by Wu and Larus [WL94], which we have implemented. For learning the loop iteration count, we collect 114 features, where 17 of them are weighted with the predicted execution frequency. The features hold characteristics of the loop structure, the loop bounds, the loop body, and the function that contains the loop. We have developed a *CoSy* engine *loopfeat* to collect these features. Internally, the feature vector is kept within a C structure named *s_loopfeat*. For learning the recursion frequency of functions, we consider 19 features. The features hold the structure of the recursive function and characteristics concerning parameters, return values, and used variables, their size and arithmetic operations on them. The feature that represents the static number of self calls of a function is weighted with its execution frequency prediction. Our *CoSy* engine *recursfeat* collects these features and keeps the feature vector in a C structure named *s_recfeat*.

To collect the features for learning execution times, we have developed a *CoSy* engine *timefeat*. We use equivalence classes of machine code instructions with respect to similar behavior to categorize instructions and we consider these categories as features. Each occurrence of a classified instruction is weighted with its execution frequency and the result is summed up to the feature that represents the category. Our engine *timefeat* provides two dif-

ferent weighting schemes. The scheme that is applicable when solely static information is available multiplies each occurrence with the static execution frequency prediction according to [WL94]. The other scheme takes the actual execution frequency derived from interprocedural path profile information as weighting factor. In total, we consider 13 categories as features. Our engine *timefeat* operates on the LIR since machine code to be emitted from LIR is only available during code generation in the back end of the compiler. The resulting feature vector is internally kept within a C structure *s_timefeat*.

In the following, we present our implementation for instrumenting the code of the program to analyze and store observed values of regarded run-time behavior. Both the feature vectors and the corresponding correct outcome of regarded behavior constitute the required training data for our supervised learning techniques.

6.2.2 Profiling

We collect path frequencies and execution times from profiling to obtain actual measured data of regarded run-time behavior. We have decided to integrate the instrumentation step into our *CoSy* compiler to preserve a continuous compilation. Thus, a user does not have to deal with interrupting the compilation because we eliminate the obligation for using external tools. Furthermore, existing dynamic binary instrumentation tools may have the drawback that they support only some *Instruction Set Architectures* (*ISAs*). For example, the instrumentation tool *Pin* [BCC+10] can cope with concurrent applications but solely supports the IA-32 and x86-64 ISAs. We present our implementation of the instrumentation in the following.

Path Profiling

To determine the actual execution frequencies of loop bodies and recursive functions, we have developed an engine *pathprof*, which automatically instruments the IR of a program. The instrumentation enables interprocedural path profiling that associates the number of times of being executed at run-time to a path in the CFG. Our engine adapts the *CoSy* engine *pprof*, which instruments the IR based on the algorithm of Ammons et al. [ABL97]. The original algorithm implemented by the engine *pprof* creates and updates arrays of counters for each function. An element in an array holds the observed execution frequency of a path in the CFG. At termination (and also during run-time for efficiency reasons if the total number of paths within a function exceeds a threshold), the observed values are written in a log file. Since we focus on mapping MPI programs to processor networks with our instantiated framework MaCAPA, we have adapted the original algorithm to cope with concurrent applications. Our adaption is *thread-safe*, i. e., it can handle shared memory implementations of the MPI standard as well. For distributed memory MPI implementations, where each process executes in its own address space, we do not have to alter the array of counters. In this case, we have adapted the

output mechanism of observed values to the effect that each process writes to a different log file. For shared memory implementations of the MPI standard, our engine *pathprof* cannot use an array of values since we would not be able to differentiate the paths of the MPI processes. In this case, we instead employ a C structure that comprises an integer value for the unique path ID and an array of integers to hold the path frequency of each MPI process separately.

The number of paths within a CFG can be quite large. Similarly, the number of MPI processes that may execute a program is not bounded. It is therefore not beneficial to create a structure for each path or to reserve an element in the array of our structure for each MPI process. Hence, we have bounded both the maximal number of paths and the maximal number of MPI processes for each path. If a number exceeds a threshold, the array elements are shared. In particular, we relate an array index to the ID (either the path ID or the process ID) that occupies the element at this index. To update an array element, we compute the index via its ID modulo the threshold and if the index is not related to the ID, the element is yet occupied. In this case, the observed values of this element are immediately written to a corresponding log file, the index is related to the new ID, and the new values are stored there.

We derive from observed path frequencies the iteration counts of loops and recursion frequencies of functions, which we present in Subsection 6.2.3. In the following, we discuss our instrumentation for measuring execution times.

Profiling Execution Time

We have developed and implemented a complete profiler for execution times within *CoSy*, which preserves a continuous compilation flow since a user does not have to use external tools. Our profiling algorithm automatically instruments the IR to analyze execution times for all functions. Additionally, the time spent for computations within the function body itself and the time spent by called functions is analyzed, as it is done by other profilers as well. We implemented the profiling algorithm in the engine *timeprof*.

We insert all functionalities, i. e., the function `gettime` that performs the measurements (see Function 4.2 on Page 89) and the start and stop of time measurements (as given by Algorithm 4.3 on Page 90), directly into the IR, which is more complicated but has several advantages. The other (simpler) choice would have been to implement the function `gettime` in a programming language and to instrument only the calls to `gettime` in the IR. As a consequence, this choice would have the following drawbacks:

1. The function must be compiled and linked with the resulting binary of each program that is compiled and mapped by our framework MaCAPA. That is, the file with the function definition must be shipped with the compiler for being present at compile time.

2. The external definition of the function in a programming language restricts the source language of the program to be compiled and mapped

by our framework since both languages must be equal (or at least compatible like, e. g., the function is written in C and the program to be compiled is written in C++).

Our choice is independent of the source language and no extra files must be given at compile time. Thus, the compiler that is extended by our framework can provide several front ends for different source languages without having to change our profiler. We have performed initial investigations on that topic in the bachelor's thesis [Gra11].

6.2.3 Path Analysis

During the *Machine Learning Phase* of our instantiated framework MaCAPA, we aim at, among others, learning the iteration counts of loops and recursion frequencies of functions. From profiling, we acquire for each path in the CFG of a function its execution frequency. We automatically derive loop iteration counts and recursion frequencies from these profiles to obtain the required runtime behavior for our learning techniques. For the former, we basically analyze the unique BB from which the loop body can be entered, called the *loop header*. For the latter, we analyze the unique *entry BB* of the corresponding CFG. We have implemented our analyses in the engine *pathana*.

The first step of both analyses is to determine the *total execution frequency* of a BB, which is the sum of the frequencies of all paths in the CFG that include this BB. That is, the total execution frequency of a BB b in a CFG *cfg* is given by

$$\texttt{totFreqBB}(b) = \sum_{p \in cfg : b \in p} \texttt{freqP}(p) \ ,$$

where `freqP` yields the execution frequency of a path p as obtained from profiling.

To analyze the loop iteration count, we start (for each nested loop) at the outermost loop. Each time a loop header is reached during execution, either the loop *body* is entered or not. Hence, the total number of executions of the loop body is the total execution frequency of the loop header minus the frequencies of all paths that include the header BB but do not include a BB of the loop body[1]. Since the function that contains the loop may be called more than once, the average loop iteration count is given by the total number of loop body executions divided by the number of function invocations. The latter is given by the total execution frequency of the unique entry BB of the CFG that contains the loop. For each nested loop, this value must be divided by the product of iteration counts of all outer loops since their iteration counts add to the total number of loop body executions. Consider a loop l in a

[1] A unique header from which the loop body is entered implies (in CoSy) a unique BB *within* the loop body that is executed at each iteration. However, we cannot use the total execution frequency of this BB to calculate the iteration count because it can be a header of a nested loop that is executed more frequently than the outer loop iterates.

CFG *cfg* at nesting level n (which is 1 for the outermost loop) with a loop header BB b_h and a set B_{body} of BBs in its loop body. Then, we calculate the iteration count of the loop l via

$$iter(l) = \frac{\texttt{totFreqBB}(b_h) - \sum_{p \in cfg : b_h \in p \wedge \neg \exists b \in B_{body} : b \in p} \texttt{freqP}(p)}{\texttt{totFreqBB}(\texttt{entryBB}(cfg)) \cdot \prod_{k=1}^{n-1} iter(l_k)} \ ,$$

where `entryBB` yields the entry BB of *cfg* and l_k is the outer loop at nesting level k.

The recursion frequency is the number of self calls of a function resulting from one external call to this function at run-time. If a recursive function is called more than once by other functions than itself, we take the average recursion frequency across these *external* calls. To analyze the total number of self calls, we sum up the frequencies of paths where a self call occurs. Since a function can call itself on a path more than once, each path frequency is multiplied with the number of self calls along the path. Then, we compute the total number of function invocations, which is given by the execution frequency of the entry BB of the CFG. The difference between this number and the number of self calls yields the number of external calls. The average recursion frequency is then given by the number of self calls divided by the number of external calls. That is, we calculate the average recursion frequency of a function f with CFG *cfg* via

$$recurs(f) = \frac{\sum_{p \in cfg : \texttt{call}(f) \in p} \{\texttt{freqP}(p) \cdot \texttt{numC}_f(p)\}}{\texttt{totFreqBB}(\texttt{entryBB}(cfg)) - \sum\limits_{\substack{p \in cfg: \\ \texttt{call}(f) \in p}} \{\texttt{freqP}(p) \cdot \texttt{numC}_f(p)\}} \ ,$$

where $\texttt{call}(f) \in p$ denotes that a call to the function f occurs in the path p and $\texttt{numC}_f(p)$ returns the number of calls to f in p.

With the analyses as described above, we obtain average loop iteration counts and recursion frequencies. Classification learning, which we use to establish predictor functions for both regarded entities, requires the behavior to be encoded as discrete values. We use the truncated decadic logarithm as classification. In other words, the number of digits within the observed value determines the class. For example, an observed value in the range between 1 and 9 is in class 1 and a value between 10 and 19 is in class 2. As we will show in Section 7.2, more than 50% of the observed values for both entities are below 10 (i. e., in class 1). Hence, our chosen classification is more fine grained in the range with the most observed values.

So far, we have described how to collect code features and profiling data from the programs used in the *Analysis Phase* of MaCAPA. We connect each feature vector with the corresponding correct outcome, which yields a table of observations per program. As a result, the training data is ready to be used for machine learning in the next phase of MaCAPA.

6.3 Machine Learning Phase

During the *Machine Learning Phase* of our instantiated framework MaCAPA, behavior predictors are constructed, which predict loop iteration counts, recursion frequencies of functions, and execution times of MPI processes. This phase is entirely implemented within the *R Project* environment [R12]. For learning iteration counts of loops and recursion frequencies of functions, we use supervised classification learning. The execution times are learned with regression modeling techniques.

The training data collected in the previous *Analysis Phase* of MaCAPA is given as a set of tables. We have implemented functions in R that load these tables into the R environment. Each row in a table represents a training example, i. e., a pair consisting of a feature vector and the corresponding correct outcome. An ML model can then be directly built from a table. We also have implemented functions in R that generate executable C code from the ML models. We present the central ideas of our learning techniques and code generation in the following.

6.3.1 Loop Iteration Count and Recursion Frequency

Our approach for learning loop iteration counts and recursion frequencies automatically generates classifiers that relate static code features to dynamic behavior. In particular, we use *Random Forest* learning [Bre01] based on classification trees. The *Random Forest* technique grows many classification trees, where at each tree only a random subset of features is used. If the training set for the current tree is drawn by sampling with replacement, about one-third of the cases are left out of the sample. This *out-of-bag* data is used to get a running unbiased estimate of the classification error as trees are added to the forest. It is also used to get estimates of variable importance. Once the forest is built, it can be applied as predictor function from features to (classified) behavior. A new observation is then classified via the majority of votes over all trees.

As described, we use 114 features for learning the loop iteration count. If the number of variables is very large, a forest can be run once with all the variables, then run again using only the most important variables from the first run. We have determined the 60 most important features that decrease the misclassification rate as follows (see [Bre01]): With every tree grown in the forest, the outcome of the out-of-bag data is predicted and the number of votes for the correct class are counted. Then, the values of a variable are randomly permuted in this data, it is also evaluated with the tree, and the number of votes for the correct class are counted. The difference between the former and the latter number, averaged over all trees, is the raw importance score for this variable. We exclusively have used these most important variables to establish the final forest. For learning the recursion frequency, we used all the 19 static code features that we have introduced in Subsection 4.2.1 (see Table 4.2). As

final classifier for both the loop iteration count and the recursion frequency, we have created 500 trees as a forest, which is the default value.

6.3.2 Execution Time

For learning execution times of MPI processes, we consider linear regression modeling, for which several learning techniques exist. We have employed the different basic learning algorithms that we have introduced in Subsection 2.3.2. Additionally, we have adapted and implemented the *Predicting Query Runtime 2* (*PQR2*) technique of Matsunaga and Fortes [MF10], which is based on a decision tree. The technique defines several model parameters (we described the original approach in Section 3.4 on Page 63). Our adaption of the PQR2 technique can be configured in the same way. At inner nodes, the *ngap* largest time gaps of the current interval are explored as possible split points, where *nskip*% of both ends of the (sorted) interval can be skipped for determining the largest gaps. The skipped *nskip* percentage of observations is used to obtain reasonable partitions of the interval. For example, if outliers are within the training data, hence having a large gap to other observations, they are located at the ends of the interval (since it is sorted). Furthermore, the minimal accuracy *minAcc* of a classifier at an inner node, the minimal number of examples *minExamples* in the interval, and the minimal interval size *minInterval* can be defined as stopping criterion for further partitioning the current range. These parameters are used since the quality of the predictor depends on the training data and if the time range is too small no additional information might be gained by further subdividing it.

We also have implemented functions to preprocess the training data and to evaluate the quality of the different techniques for our implementation module *train_time* within the R environment. We have investigated this topic in the master's thesis [Voi13]. When observations in the training set contain noisy features or when the feature scales are not consistent with their importance, the accuracy of ML algorithms can be degraded. For example, the *k-Nearest Neighbor* (*k-NN*) algorithm using the *Euclidean distance* for determining the k nearest neighbors is sensitive to the local structure of the data. If one of the features has a broad range of values, the distance will be governed by this particular feature. Hence, it can be useful to scale features in order to make the features independent of each other. We normalize the features via rescaling the range of each feature to [0, 1]. That is, we replace each value x of a feature by its normalized value x' given by

$$x' = \frac{x - min}{max - min} \ ,$$

where min is the minimal value of this feature in all observations and max is the maximal value.

6.3.3 Generation of Executable Code from the ML Models

The final step of the *Machine Learning Phase* of our instantiated framework MaCAPA is the code generation from our ML models to obtain executable predictors. Within R, new data can be predicted directly with a given model. Since we employ predictions about unknown run-time information within the *CoSy* compiler framework (where the engines are written in C), we automatically translate the representations of the used models into C code, which is then automatically integrated into *CoSy*. We have implemented a module *code_gen* within the R environment that performs this code generation.

For learning loop iteration counts and recursion frequencies of functions, we have considered *Random Forest* learning, which is based on classification trees. For each tree, the inner *decision nodes* split the input space in two subspaces and the *leaf nodes* define the classes. The decision at an inner node m is based on a comparison $f_m(\mathbf{x})\colon x_j \geq w_m$ where x_j is a dimension of the feature vector $\mathbf{x}$ and w_m is a threshold value. For each inner node, we create an `if-then-else` construct that checks this condition. We start at the root of each tree and we proceed recursively with the children of each node. For the left child, we generate C code within the `then`-branch and for the right child, we take the `else`-branch. Whenever we reach a leaf node, the generated C code returns the annotated class as prediction result of the current tree. Since the *Random Forest* technique creates many classification trees, we automatically generate code that adds up the votes of all trees for the individual classes and returns the class with the majority of votes as final decision for a given feature vector. The code is enclosed within a function that takes the internal representations of the feature vectors. That is, it takes the C structure named *s_loopfeat* for learning loop iteration counts and the C structure named *s_recfeat* for learning recursion frequencies.

For learning execution times, we have adapted and implemented the PQR2 technique. At inner nodes of our PQR2 tree, the classifier with the highest accuracy of a pool of possible classifiers predicts the two subranges of the current range (i. e., the threshold) for the children. We have used the *k-NN* technique, *classification trees*, and *Naïve Bayes* as possible classifiers at inner nodes. At leaf nodes of the tree, the best regression method from a pool of basic regression algorithms determines the outcome. Our implementation of the PQR2 tree uses the *Ordinary Least Squares Estimation* (*OLS*) technique and *Support Vector Machines* (*SVMs*) as possible regression methods. We have developed a code generator in R, which takes a PQR2 tree and automatically generates C code from it. At each node of the PQR2 tree, we proceed similarly to our approach for the classification tree as described above, starting at the root of the tree. Especially, we generate C code for the respective ML technique at each node that implements the corresponding predictor function. For the k-NN and the SVM techniques, which explicitly use the training data to derive a prediction, we first generate code to store the whole training data. In particular, we store the feature vectors of the training data as an array of structures *s_timefeat* and an array that holds the related execution times. These techniques may be used at several inner nodes. For efficiency reasons, we

store the training data only once and generate code that holds which elements in the arrays are relevant[2] together with their corresponding outcome at the current node. The generated C function that implements the PQR2 tree takes the internal representation of the feature vector for learning execution times as argument, i. e., the C structure *s_timefeat*. Since we scaled the features for learning the ML models as described in the preceding subsection, we also generate C code that scales the features of the input vector accordingly.

As a result of the implemented *Training Phase* of MaCAPA, we obtain executable predictors for loop iteration counts, recursion frequencies, and execution times. We use these predictors in the *Application Phase* of MaCAPA to determine the communication overhead and the computational costs of MPI processes. We describe peculiarities of our implementation of the *Application Phase* in the next section.

6.4 Mapping Phase

In the *Training Phase* of MaCAPA, predictor functions for regarded run-time behavior are automatically generated. The functions are given in C and use the data structures that were defined by our *CoSy* engines for the analyses of code features. Hence, the predictors can be used in the compiler to predict the regarded behavior during the *Application Phase* of MaCAPA. This only requires an additional analysis step of static code features, which is done by our respective engines *loopfeat*, *recursfeat*, and *timefeat*.

The basis of the *Application Phase* is the analysis of the behavior of MPI processes. From this, our mapping algorithm allocates MPI processes to PEs of the processor network, guided by our cost model. We briefly present the implementation of our analysis and the mapping in the following.

6.4.1 Analysis of MPI Processes

Input to our MaCAPA framework is the source code of an MPI program, which first is transformed by the front end of the *CoSy* compiler into CCMIR. Our analyses and our mapping heuristics then work on this IR. The *CoSy* compiler cannot process more that one source file at the same time. We therefore have stored our intermediate analysis results for each source file in log files and read collected information from these files[3]. The information we have stored includes the *Call Graph* (*CG*) of the current program together with execution frequencies of function calls, the information which MPI processes can take a conditional path in the CFGs of functions, and the results of our communication analysis. We describe notable aspects of our implementation in the following. We start with the analysis of executed instructions and of syn-

[2]The predictors at each node were established only with the elements that reach this node.
[3]If the *CoSy* compiler processed all source files at once, we could store the intermediate results directly in the CCMIR.

chronization between MPI processes that is independent from communication. Then, we continue with our communication analysis.

Executed Instructions and Synchronization

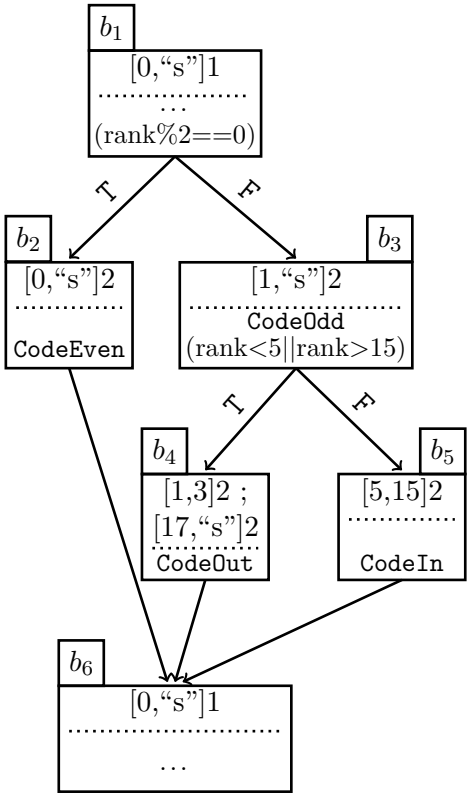
We have implemented the algorithm to derive the ID dependent behavior of MPI processes, which we have defined in Algorithm 4.4 for the general MaCAPA framework. Instead of generating an ID dependent CFG for each MPI process separately, we have annotated the given CFGs with the information which MPI process can reach which BB. We have assigned to each BB a list of intervals. Each interval constitutes a range of IDs, which represents the processes that can enter the BB during execution of the program. Since not all values of interval bounds can statically be derived, our implementation additionally defines *symbolic bounds* in these cases (represented by strings). We also associate a *step* i with each interval, which represents that only every i'th ID is within the range. This enables us to cope with guards of conditional paths that use modulo operations to check whether a rank of a process has a certain value. Our implemented algorithm can handle all other basic arithmetic operators $(+, -, \cdot, /)$ and all relational operators $(<, \leq, =, \neq, \geq, >)$ as well.

We give an exemplary MPI code snippet in Figure 6.2a for which our implemented algorithm automatically analyzes the exact behavior. At Line 6 of the code snippet, all processes obtain their respective rank and execute the condition at Line 7. This condition checks whether the rank modulo two equals zero. Hence, only each second process is allowed to execute `CodeEven` at Line 9, beginning with the process having rank zero. Likewise only each other second process is allowed to execute `CodeOdd` at Line 12, beginning with the process having rank one. At Line 13, the condition checks whether the rank is outside the interval $[5, 15]$. If the condition is fulfilled, the corresponding processes can execute `CodeOut` (Line 15), and otherwise, they execute `CodeIn` (Line 18). Finally, all processes execute the code at Line 21.

Our analysis result is shown in Figure 6.2b. The large rectangles represent BBs, which are labeled $b_1, b_2, \ldots, b_6$ within small rectangles. The text above the dotted line within each rectangle represents the (list of) intervals, where $[lb, ub]i$ represents the interval $[lb, ub]$ with step i. At BB b_4, the list has two intervals and for the other BBs, our analysis yields lists with only one element. Below the dotted lines are the code snippets. The bottom text in b_1 and b_3 are the conditions at Line 7 and Line 13, respectively. If they are fulfilled, control flow passes through the left `T` edge, otherwise through the right `F` edge. We have defined a symbolic upper bound "s" that represents the total number of MPI processes executing the program. Since all processes can enter the BB b_1 (which includes the code of Lines 4 to 7), its interval is $[0, "s"]1$. The analyzed intervals of b_2 and b_3 exactly reflect the outcome of the condition in b_1. Similarly, the intervals of b_4 and b_5 exactly reflect the condition in b_3. Note that our analysis correctly determines that processes with ranks 4 or 16 cannot enter the BB b_4. Our analysis automatically merges the intervals of the

```
1  int main(int argc, char *argv)
2  {
3    int rank, size;
4    ...
5    /* get current process id */
6    MPI_Comm_rank(..., &rank);
7    if(rank % 2 == 0){
8      /* even ranks allowed */
9      CodeEven
10   }else{
11     /* odd ranks allowed ... */
12     CodeOdd
13     if(rank < 5 || rank > 15){
14       /* ... and not [5,15] */
15       CodeOut
16     }else{
17       /* ... and only in [5,15] */
18       CodeIn
19     }
20   }
21   ...
22 }
```

(a) MPI code fragment



(b) Corresponding annotated CFG

Figure 6.2: Conditional execution by MPI processes

lists at join points of control flow whenever possible. As a result, the analyzed interval of BB b_6 precisely denotes that all processes can enter this BB. We store the CG and the CFGs together with our analysis result in a log file using a *Graph Description Language* such that it can be visualized and interactively explored (for example with the graph visualization tool *aiSee* [AI13]). The presented approach is based on our initial investigations in the bachelor's thesis [BT12].

We have implemented the analysis of synchronization between MPI processes, which we have defined in Subsection 5.3.3. Both the analysis of executed instructions and the synchronization that is independently from communication are implemented in the *CoSy* engine *taskana*. As synchronization, we consider barriers in the following cases:

1. The barrier is in a BB that post-dominates the entry BB of the CFG of MPI processes (i. e., the entry BB of CFG cfg_{main} of the `main` function).

2. The barrier is in a BB that post-dominates the entry BB of the CFG of a called function, where the function call must be in a BB that post-dominates the entry BB of the `main` function.

Our implementation currently handles the latter case only for functions that are defined in the same source file as the `main` function (since the *CoSy* front

end can only process one source at the same time). Based on analyzed barriers, we determine the execution phases of the program, which we annotate to the BBs. We use this information to analyze the ordering constraint on the execution of MPI processes, which we derive from the unique dominance order of the barriers, and to limit the set of possible communication partners (if the exact partners cannot be determined).

Communication Analysis

We have implemented the analysis of communication partners and the communication volume (see Subsection 5.3.2) in the *CoSy* engine *commana*. To determine the communication partners as accurately as possible, it is necessary to analyze environmental variables such as the respective ranks of the sender and the receiver or the *tag* of message calls that connects a send operation with a receive operation. To that end, we have implemented a function `DUchain` based on a *Reaching Definition Analysis*. It determines the values that environmental variables may have at each BB of the CFGs. We have investigated the analysis of possible communication partners in the bachelor's thesis [Ger13]. The current implementation does not support the dynamic creation of groups. As a consequence, we safely over-approximate the set of possible communication partners of collective MPI operations with the set of all processes that execute the program.

For our analysis of the total communication volume between MPI processes, we employ machine learned predictors, which were established during the *Training Phase* of our instantiated framework MaCAPA. Based on these predictors, *loopfeat* generates predictions for loop iteration counts and *recursfeat* generates predictions for recursion frequencies. Both are used to predict communication overheads as defined in Subsection 4.4.1 for our general framework. If MPI processes communicate in synchronous mode, we tag the corresponding communication with this information.

Based on our analyses presented above and on our established predictor functions for unknown run-time behavior from the *Training Phase* of MaCAPA, we establish a cost model, which guides our mapping heuristics. We convey the central ideas of our mapping approach in the following.

6.4.2 Mapping MPI Processes based on Cost Model

In this subsection, we present our implementation of the final step of our instantiated MaCAPA framework, the mapping of MPI programs to the target processor network. We have implemented our cost model that guides the mapping heuristics and our mapping algorithm within the *CoSy* engine *taskmap*. First, our implementation module *timefeat* generates predictions for the execution times on different PEs and for the best performing PE. The best performing PE is determined with the proposed *basic* variant for learning the PE that executes an MPI process most efficiently (see Subsection 4.3.1). That

is, it is given by the PE causing the lowest (predicted) execution time. These predictions together with the analyzed executed instructions and predicted communication overheads establish our cost model. Since the engine *timefeat* operates on the LIR in the back end of the compiler, our engine *taskmap* is also located in the back end.

We have implemented our developed algorithm for the mapping of MPI processes to the PEs in a processor network, which we have defined for the general framework. We initially have investigated that topic in the master's thesis [Baa11] and have developed the full version based on this experience. Our implementation first assigns to each synchronization point s the expected point in time when it will be reached during execution ($exp(s)$, see Algorithm 4.5) and whether this point in time is postponable without performance degradation ($post(s)$). It also assigns to each part of an MPI process a priority, which is based on its average predicted execution times on PEs and on average predicted communication times w. r. t. different allocations of communication partners. For efficiency reasons, we compute necessary information only once and store this information in (high-dimensional) arrays since elements in these arrays can be accessed in constant time. For example, the communication times are stored in an array where an element represents for each process, for each of its parts, and for each communication partner the time when both partners are allocated to particular PEs, which results in a five-dimensional array. We then allocate the MPI processes to the PEs of the processor network as defined in Algorithm 4.5. Finally, the derived mapping is stored in a file, which is used by the MPI run-time daemon to start the processes at the corresponding PEs.

We have implemented the automatic instrumentation of the CCMIR[4] to signal the MPI run-time daemon for reallocating the processes at synchronization points. In particular, we send the POSIX reliable signal `SIGUSR1` to the MPI run-time daemon. We also have implemented the functionality within the MPI run-time environment that catches the signal. The signal is automatically forwarded to run-time daemons of remote CPUs in the processor network, where it is also caught by the respective daemons. Our current implementation solely considers the dynamic reallocation of MPI processes between PEs of the same CPU (i. e., the cores of the CPU). It does not consider the dynamic reallocation between different CPUs in the processor network since this would involve to save execution states of all processes, to safely stop a process, and to start it after restoring the current state at the new CPU. Furthermore, the binary and computed data such as dynamically allocated data in the heap space would have be transferred to the new CPU. We therefore set the reallocation cost between different CPUs to infinite and the reallocation cost between PEs of the same CPU to zero (since the cost for moving processes between cores is negligible). For the reallocation, we have used the function `hwloc_set_proc_cpubind` from the *Portable Hardware Locality* software package.

[4] In the back end of *CoSy*, both the CCMIR and the LIR are available.

In this section, we have presented our implementation of the *Application Phase* of our instantiated MaCAPA framework where MPI processes are automatically mapped to a given heterogeneous processor network. In the following, we summarize our entire implementation.

6.5 Summary

In this chapter, we described how we have implemented the instantiated framework MaCAPA, which aims at improving the mapping of MPI programs to heterogeneous networks of processors. In Table 6.1, we list the number of *Source Lines of Code* (*SLoC*) for each implemented module (comments and empty lines were not counted). It also shows the language used for our implementation (column *Lang.*). The *Analysis Phase* is implemented using *CoSy* for the collection of code features, for the instrumentation of the IR to analyze and store considered run-time behavior during profiling, and for the extraction of loop iteration counts as well as recursion frequencies from observed path frequencies. The *Machine Learning Phase* of MaCAPA is entirely implemented within the *R-Project* environment for statistical computing. This phase yields executable predictors, given in C, to predict loop iteration counts, recursion frequencies, and execution times based on static code features. During the *Application Phase* of MaCAPA, which is fully implemented within *CoSy*, we analyze the possible behavior of MPI processes, i. e., executed instructions by MPI processes as well as the communication and synchronization between processes. Based on this, we establish a precise cost model with the help of the learned predictors for unknown run-time behavior. Using this cost model, the processes are automatically mapped to the PEs of the given heterogeneous network of processors. As a result, we obtain a complete experimental platform, which supports the full C language and the complete MPI 3.0 standard [MPI12]. In the next chapter, we evaluate our implemented framework. First, we demonstrate the accuracy of our ML based run-time behavior predictors. Then, we investigate the performance gain for whole programs that our framework achieves.

Table 6.1: Source Lines of Code (SLoC) of implementation modules

SLoC	Lang.	Module
1303	C	*CoSy – loopfeat*: feature analysis for loops
2102	C	*CoSy – recursfeat*: feature analysis for functions
1785	C	*CoSy – timefeat*: feature analysis for processes
654	C	*CoSy – pathprof*: instrumentation (path frequencies)
2450	C	*CoSy – timeprof*: instrumentation (time measurements)
920	C	*CoSy – pathana*: extracting loop iteration counts and recursion frequencies from path frequencies
2382	R	*R-Project – train_freq*: learning loop iteration counts and recursion frequencies, experimental evaluation
3587	R	*R-Project – train_time*: learning execution times, experimental evaluation
1647	R	*R-Project – code_gen*: generation of executable predictors
5066	C	*CoSy – taskana*: analysis (executed code, synchronization)
6437	C	*CoSy – commana*: analysis (communication)
1579	C	*CoSy – taskmap*: mapping of processes based on cost model
626	C	*MPI run-time environment*: signal handling, reallocation of processes
30538		Total

7 Experimental Results

In this chapter, we describe the results of our experiments with the implementation of our framework MaCAPA, instantiated for the mapping of *Message Passing Interface* (*MPI*) programs to heterogeneous processor networks. The instantiated framework employs a *Machine Learning* (*ML*) based cost model to rate the gain of alternative mappings. To acquire the necessary training data for the ML techniques and to evaluate the performance gain of MaCAPA, we have established a collection of benchmarks, which we present next. Then, we discuss the evaluation of our ML based predictors for regarded run-time behavior in Section 7.2. Afterwards, we investigate the performance gain that our instantiated framework achieves in Section 7.3.

7.1 Representative Program Suites

We evaluate the performance improvement of our framework with several MPI programs. The set of available concurrent MPI programs, however, comprises far from sufficiently enough observations for the training phase. We therefore have trained the ML models with a large number of sequential programs, which were taken from various benchmark suites[1]. In the following, we present our collection of sequential benchmarks. In Subsection 7.1.2, we present the MPI benchmarks that we have used for the evaluation of program performance.

7.1.1 Sequential Benchmark Suites

The ML algorithms employed by MaCAPA require feature extraction and profiling from a comprehensive suite of programs. Having a large set of observa-

[1]It is sufficient to use sequential benchmarks for learning. Loop iteration counts and recursion frequencies of functions are not affected by parallel versus sequential execution. Concerning execution times and the best performing PE, which we solely consider for predicting computational (and not communication) overheads, the ML models exactly yield the relevant information – the cost for executing sequential code without the cost for communication.

Table 7.1: Serial benchmark suites

No.	Benchmark suite	Iteration count		Recursion freq.		Time	
		#Progs.	#Loops	#Progs.	#Fcts.	#Progs.	#Obs.
1	BioBench [AJW+05]	6	750	4	16	1	63
2	Bit Stream [Gus10]	5	17			4	84
3	cBench [Fur09]	27	1976	7	7	18	14313
4	CSiBE [BC04]	23	1312	4	19	12	1819
5	Fhourstones [Tro10]	1	14	1			
6	FreeBench [RW02]	7	208	2	3	3	134
7	GCbench [Boe10]	1	4	1		1	30
8	Heaplayers [BZM01]	7	802	5	51	3	402
9	LLCbench [ML98]	2	27			2	115
10	LLVM [CDSL11]	63	3269	16	105	37	1115
11	LMbench [MS96]	24	1107	2	3		
12	MallocBench [GZH93]	6	573	5	34		
13	McCat [HP98]	9	76	6	14	3	153
14	MediaBench [LPMS97]	4	55			3	48
15	MediaBench II [FSTW09]	11	1897	3	8		
16	MiBench [GRE+01]	17	364	3	3	12	436
17	NPB [dW02]	8	1829			8	594
18	OldenBench [CR95]	10	116	8	27		
19	Prolangs-C [LR97]	19	519	8	13	6	348
20	Ptrdist [ACP+95]	5	202	3	7	1	26
21	Shoot [Ful04]	20	100	4	6	15	186
22	SPEC CPU95 [Rei96]	38	1861	4	12	1	49
23	SPEC CPU2000 [Hen00]	38	2761	5	45	5	1262
24	SPEC CPU2006 [Hen06]	38	2949	6	35	3	1077
25	Splash2 [WOT+95]	12	1000	3	9	8	303
26	SWEET WCET [Gus07]	30	271	1	1	29	232
27	Trimaran [RC07]	7	44	1	2	6	108
28	UnixBench [Smi07]	16	55	1	1	1	21
29	Versabench [RBAA04]	9	93	1	3	8	119
30	X Bench [AS04]	3	629				
	$\sum$	391	24880	104	424	190	23037

tions as training data is crucial for the ML techniques since they generalize the behavior from the training samples. Furthermore, we require that the ML based predictors are applicable to all kinds of programs. Hence, the training data should cover a wide range of behavior. To that end, we have established a huge benchmark collection that comprises 391 sequential programs from 30 benchmark suites (see Table 7.1). Our collection encompasses various real-world programs from different application domains. We have used

these benchmarks for learning loop iteration counts, recursion frequencies of functions, and execution times.

For learning loop iteration counts, we have analyzed and profiled the 391 programs (see column *Iteration count – #Progs.*). We have analyzed in total 24880 loops, which were called at run-time of the program at least once (column *Iteration count – #Loops*). We are only able to learn iteration counts of loops that were called at least once because the actual iteration count must be known and a loop that was not called does not iterate.

For learning the recursion frequency, we have found recursive functions in 104 out of 391 benchmarks (see column *Recursion freq. – #Progs.*). We have analyzed and profiled 995 recursive functions in total. As input, we have used the standard data provided by the benchmark suites. For example, we have used the *test*, *train*, and *ref* data sets for all SPEC benchmarks. At run-time, only 424 of the recursive functions actually were called at least once by another function (see column *Recursion freq. – #Fcts.*). This is our database for learning because recursion frequencies cannot arise at a function that was not called.

We have learned execution times of functions with observations from different *Central Processing Units* (*CPUs*). This establishes a different ML model for each CPU. We have employed the resulting predictors to rate the CPUs according to their execution performance. However, several programs do not execute correctly on at least one CPU[2] or do not compile, e. g., due to library dependencies on a CPU. To be able to compare execution time predictions for all CPUs, we have taken only programs that compile and execute correctly on every CPU. As a result, we have learned execution times from 23037 observations in 190 benchmarks (see columns *Time – #Obs.* and *– #Progs.*, respectively).

7.1.2 Parallel Benchmark Suites

We have established a collection of MPI benchmarks to assess the resulting run-time performance of MPI programs when they are mapped with our framework to a processor network. The programs are written in C using MPI for parallel processing. The benchmarks mainly are taken from two benchmark suites. In Table 7.2, we show the collected benchmarks. Most well known is the *SPEC MPI2007* [MvWL+10] suite from the *Standard Performance Evaluation Corporation* (*SPEC*, from which already several sequential benchmarks were taken). The benchmark programs are natively developed for MPI-parallel end-user applications, as opposed to being synthetic benchmarks or even parallelized versions of sequential benchmarks. The suite contains 18 benchmarks in total, where 4 of them are written in C and the others are written in Fortran90, a mixture of C and Fortran90, or C++. Unfortunately, the CoSy compiler we used for our framework does not provide a front end for Fortran90 or C++ and

[2]For example, several programs yield segmentation faults during execution on one CPU, though executing correctly on other CPUs.

Table 7.2: MPI benchmark suites

No.	Benchmark suite	#Progs.	Description
1	SPEC MPI2007	4	Industry-standardized benchmarks
2	Cbench	25	Scalable Cluster Benchmarking
3	IOR	1	HPC benchmark, version 2.10.3
4	LANL-HPC-5	1	Benchmark for parallel file systems
5	PIO	2	Benchmarks for parallel I/O systems
6	RAxML-7.0.4	1	Bioinformatics application
	$\sum$	34	

an existing tool for translating Fortran to C (*f2c*, Feldman et al. [FGMS91]) only supports Fortran77. Hence, we took the 4 benchmarks that are written in C, namely *milc* for doing simulations to study quantum chromodynamics, *RAxML* for computing large phylogenetic trees from alignments of DNA sequence data, *tachyon* for parallel ray tracing, and *dmilc*, a modified version of *milc* to exploit higher degrees of parallelism.

The second benchmark suite is *Cbench* [SBS13] for scalable cluster benchmarking. This large suite includes many of the common open source tools used in Linux clusters such as the *Intel MPI Benchmarks* (the newest version of the venerable Pallas benchmarks), the *ASC Sequoia Benchmarks*, and the *Perftest suite* of tests from MPICH. Additionally, we have collected five further MPI benchmarks. The *High Performance Computing* (*HPC*) benchmark *IOR* [LMM13] can be used for *Interleaved-Or-Random* performance testing of parallel file systems using various interfaces and access patterns. The *LANL-HPC-5 MPI-FTW* [GT13] benchmark is used to traverse a directory tree using MPI. The *PIO* suite [PIO13] provides two benchmarks for use in characterizing the performance of a parallel I/O system. The bioinformatic application *RAxML* [SLM04] is one of the most popular applications for phylogenetic analysis. It is the older version 7.0.4 of the program as in the SPEC MPI2007 suite (to which the RAxML-VI-HPC version [Sta06] was added in 2010) with different data sets[3].

The used MPI benchmarks confirm that our implemented framework can cope with huge software in the area of HPC like, e. g., the bioinformatic application *RAxML*. Furthermore, it demonstrates that our framework is applicable to programs of real-world application domains.

[3]The input sets are the data used for the paper [OZSA07].

7.2 Evaluation of the Predictors

In this section, we investigate the accuracy of our ML based predictors for unknown run-time behavior. Our ML techniques target four goals, namely learning iteration counts of loops, recursion frequencies and execution times of functions, and the *Processing Element* (*PE*) that executes a function most efficiently. We present the evaluation of our experiments for learning loop iteration counts in Subsection 7.2.1, for learning recursion frequencies in Subsection 7.2.2, for learning execution times in Subsection 7.2.3, and for learning the best performing PEs in Subsection 7.2.4.

7.2.1 Prediction of Loop Iteration Count

For our experiments, we have used 391 programs from 30 benchmark suites (see Table 7.1). As learning technique, we have considered *Random Forest* learning [Bre01]. We have applied our approach to 24880 loops and we have observed iteration counts ranging from one to over four billion. As classification of loops, we have used the truncated decadic logarithm of the loop iteration count[4]. In other words, the number of digits within the iteration count determines the class, thus leading to 10 classes. As precision measure, we have used the average deviation between predictions and correct classes (*mean absolute error*). Additionally, the *Pearson's product-moment correlation coefficient* [LRN88] between predictions and correct classes indicates whether a relationship was actually learned.

In this subsection, we first present our experimental results for learning loop iteration counts. To interpret the results, we afterwards compare our approach against other heuristic results. With that, we can identify the amount of information that was extracted by ML.

Results from Cross Validation

To determine the general applicability of classification learning to predict loop iteration counts, we first have performed *self-validation*. That is, we have applied the predictors to the data it was trained with. For realistic results, we have applied the learned predictors to unseen data for evaluation (i. e., the benchmarks that the predictors where trained with and the benchmarks used for evaluation are disjoint). In particular, we have performed *leave-one-program-out cross-validation* for assessing how our results will generalize to an independent data set. That is, we remove all loops of a certain program from the training set, train the predictors with the remaining set, and predict the removed loops with the established predictors. This cross-validation is iteratively performed such that each program is left out once, which results

[4]Considering the distribution of iteration counts (compare Figure 7.3 on Page 172), our chosen classification is more precise exactly there where most loops have their iteration counts.

in 391 experiments. Our results shown for cross-validation are the average over all experiments.

In Figure 7.1, we depict the observed mean absolute errors and correlations with our ML based technique and with other basic approaches (which we discuss later). The mean absolute error for self-validation, which is shown in Figure 7.1a labeled *self*, is 0.02 (with a standard deviation of 0.20, indicated by bars) and the correlation is very high (0.99, see Figure 7.1b). This gives a lower bound on the precision of predictions. The error for cross-validation (*val*) is slightly higher than for self-validation (0.72, with a standard deviation of 1.14). The correlation decreases to 0.47, which nevertheless indicates that a (moderate) relationship could be learned. For a closer view on the errors, the Δk-accuracy can be examined. It denotes the fraction of predictions with a maximum absolute deviation of k classes. We show the Δk-accuracy in Figure 7.2, where the mean absolute error is indicated by the abscissa of the cross. For the *self* predictions, about 98% are correctly predicted and more than 99% maximally deviate by one class. Our evaluation with unseen data (*val*) yields almost the same accuracy: 67% of the predictions are correct, and more than 83% have an error of at most one class.

In summary, having 83% of the predictions with a maximal deviation of one class is well-suited for our optimization. To assess the accuracy of our approach, we next compare our results with other heuristic predictions of loop iteration counts.

Comparison Against Other Heuristics

We model several heuristics to compare our predictions against the precision of other basic approaches. For example, a purely static analysis simply chooses a fixed value as iteration count or selects a value at random.

In Figure 7.3, the distribution of the classified loop iteration count is shown. As can be seen, more than 50% of the loops actually iterate at most 9 times (i. e., their iteration counts are classified into class 1). Based on this observation, it is clear that always predicting class 1 will give the best result when a heuristic simply chooses a fixed value. Therefore, we have performed the corresponding experiment. Figure 7.1 shows its result, labeled *prd.1*. The mean absolute error is 0.98 and, of course, the correlation is 0.00. The Δk-accuracy (see Figure 7.2) reflects the distribution of the iteration counts. More than 50% of the predictions are correct, about 76% maximally deviate by one class, and about 87% have an error of at most two classes.

Considering nested loops, a common heuristics for choosing a fixed value as iteration count (as used in the commercial CoSy compiler) is to take the value 10 for the outermost loop. For each nesting depth, the predicted execution frequency of the loop is multiplied by 10. That is, each loop is predicted to iterate 10 times. Hence, the iteration count of each loop would be in class 2 by using this heuristics. We also have modeled the case that each loop iterates 100 times, i. e., the heuristics always predicts the iteration count to be in class 3. As

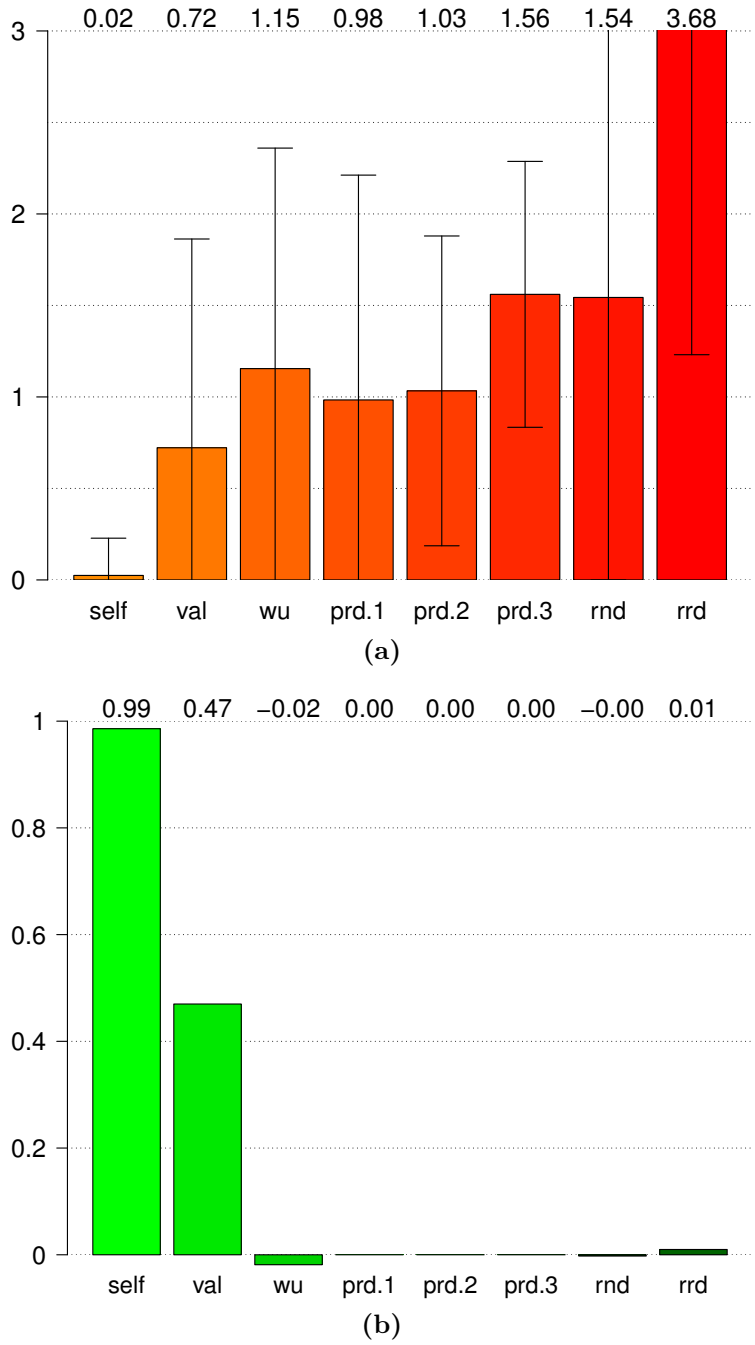


Figure 7.1: Loop – **(a)** mean absolute error and **(b)** correlation

expected regarding the distribution, both heuristics perform even worse than predicting class 1 for each loop. Always predicting class 2 (seeFigure 7.1a, label *prd.2*) and always predicting class 3 (label *prd.3*) yield a mean absolute error of 1.03 and 1.56, respectively. In all cases, we perform better than simply choosing a fixed value for the loop iteration count. This is demonstrated by our experimental results showing a lower mean absolute error and a good correlation between predictions and the actual class.

A non-trivial heuristics for predicting loop iteration counts without profiling is the approach of Wu and Larus [WL94] for statically predicting execution

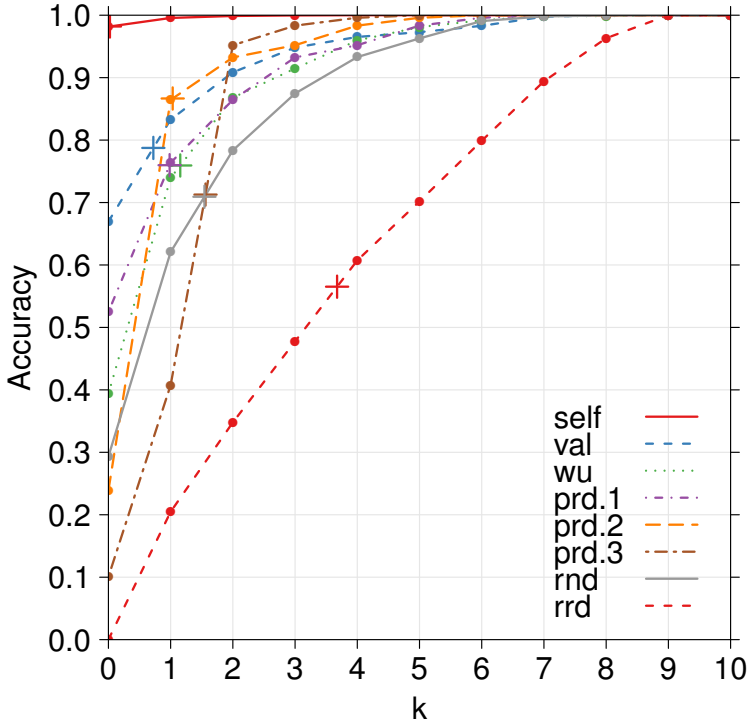


Figure 7.2: Loop – Δk-accuracy

frequencies of BBs within the CFG. Instead of choosing a fixed value, for each loop a certain execution frequency is predicted. The result of the corresponding experiment is shown in Figure 7.1, labeled *wu*. The predictions deviate by 1.15 classes on average from the actual class and the correlation is -0.02, i. e., no relationship exists between predicted and actual classes. Again, we perform

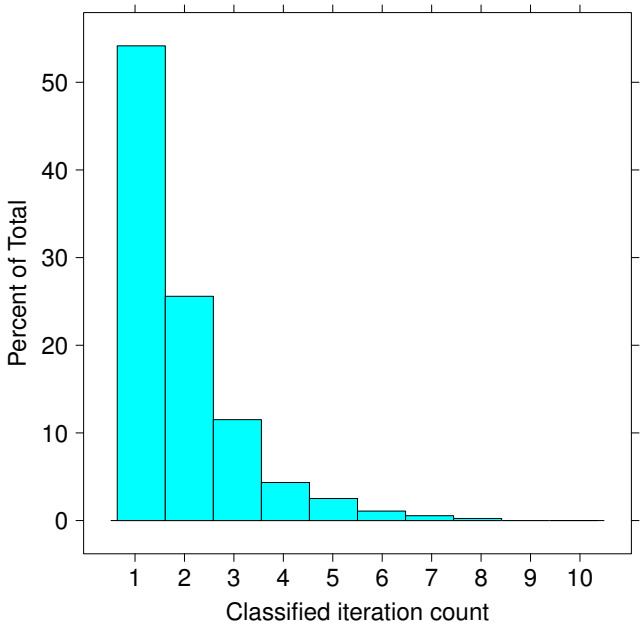


Figure 7.3: Distribution of classified loop iteration counts

better as is demonstrated by our lower mean absolute error and the higher correlation.

To model the heuristics that selects a value at random for the iteration count, we use two random predictors: a pure random prediction of one of the classes and a prediction that first determines the distribution of the classes of the training set and then predicts a class based on that distribution. To prevent having outliers as result, we have repeated the corresponding experiments 50 times and have taken the average. The results of random prediction based on the distribution of classes is labeled *rnd* in Figure 7.1. The mean absolute error is 1.54 with a standard deviation of 1.54, and the correlation is 0.00. The pure random predictor yields the worst result (see Figure 7.1, label *rrd*). The mean absolute error is 3.68 with a standard deviation of 2.44 and the correlation is 0.01. Our ML based approach yields much lower errors than the random prediction.

In summary, we have shown in this section that other heuristics for predicting the loop iteration count perform inferior to our approach. This gives evidence that our learning algorithm can successfully identify the relationship between features and classes for a given program. Since we have used a wide range of programs with different behavior, our results show that the predictions are accurate across different domains.

7.2.2 Prediction of Recursion Frequency

For learning the recursion frequency, we have identified 424 functions that were called at least once by another function at run-time (see column *Recursion freq. – #Fcts.* of Table 7.1). Since we have used *Random Forest* classification learning, the run-time behavior has to be discretized into classes. As classification, we have used the truncated decadic logarithm of the recursion frequency. We have observed recursion frequencies up to 300 million, which leads to 9 classes[5]. As precision measures, we again have used the average deviation between predictions and correct classes (*mean absolute error*) and the *correlation* between predictions and correct classes.

In this subsection, we first present our experimental results for learning recursion frequencies. Afterwards, we compare our approach against other heuristic results to identify the amount of information that was extracted by ML.

Results from Cross Validation

As with the experiments for learning loop iteration counts, we first have applied the predictors to the data it was trained with, which gives a lower bound on the precision of predictions. For realistic results, we then have applied the

[5]The highest recursion frequency stems from the *Ackermann function* within the Shoot benchmark suite.

predictors to unseen data for validation using leave-one-program-out cross-validation.

Our experimental results demonstrate the accuracy of our approach very well. We show the mean absolute errors and the correlations in Figure 7.4. The mean absolute error for *self*-evaluation is 0.04 (with a standard deviation of 0.25, indicated by bars) and the correlation is with 0.99 nearly perfect. For cross-validation (*val*), the error is higher than for self-evaluation (0.72, with a standard deviation of 1.33), but still, we have a high correlation (0.45). For a closer view, the Δk-accuracy is shown in Figure 7.5, which denotes the fraction of predictions with a maximum absolute deviation of k classes. The mean absolute error is again indicated by the abscissa of the cross. For the *self* predictions, nearly 98% are correctly predicted, and 99% maximally deviate by one class. Our validation also yields a high accuracy: about 65% of the predictions are exactly correct, above 81% have an error of at most one class, and about 89% have an error of at most two classes.

In summary, with more than 81% of the predictions that maximally deviate by one class, our learning technique is suitable for our aims. To further evaluate our approach, we next compare our results against other heuristic predictions of the recursion frequency.

Comparison Against Other Heuristics

For comparing our predictions against other heuristics, we have performed experiments with different basic approaches. For example, a purely static analysis can ignore the fact that the function is recursive and, hence, treat the function as though it was called once. Using this heuristics, the function is classified into class 1. A more precise static analysis takes the static recursive call count (i. e., the number of occurrences of self calls in its body) as fixed value. Assuming real applications, we state that this static call count is maximally 100, which results in classifying the function into class 1, 2, or 3.

The distribution of the classified recursion frequency is shown in Figure 7.6. Based on the observation that more than 50% of the recursion frequencies are classified into class 1, it is clear that always predicting this class will yield the most precise result when choosing a fixed value. Figure 7.4 shows the result of the corresponding experiment, labeled *prd.1*. The mean absolute error is 1.04 with a standard deviation of 1.50, and, of course, the correlation is 0.00. The Δk-accuracy (see Figure 7.5) reflects the distribution of the recursion frequency. More than 50% of the predictions are correct, about 75% maximally deviate by one class, and about 88% have an error of at most two classes. As expected regarding the distribution, always predicting class 2 (see Figure 7.4, label *prd.2*) and always predicting class 3 (label *prd.3*) performs even worse. The prediction of class 2 yields a mean absolute error of 1.36, with a standard deviation of 0.90, and the correlation is 0.00. For always predicting class 3, the error increases to 1.61, with a standard deviation of 0.78, and the correlation is 0.00. We perform better than simply choosing a fixed value for the recursion frequency in all cases.

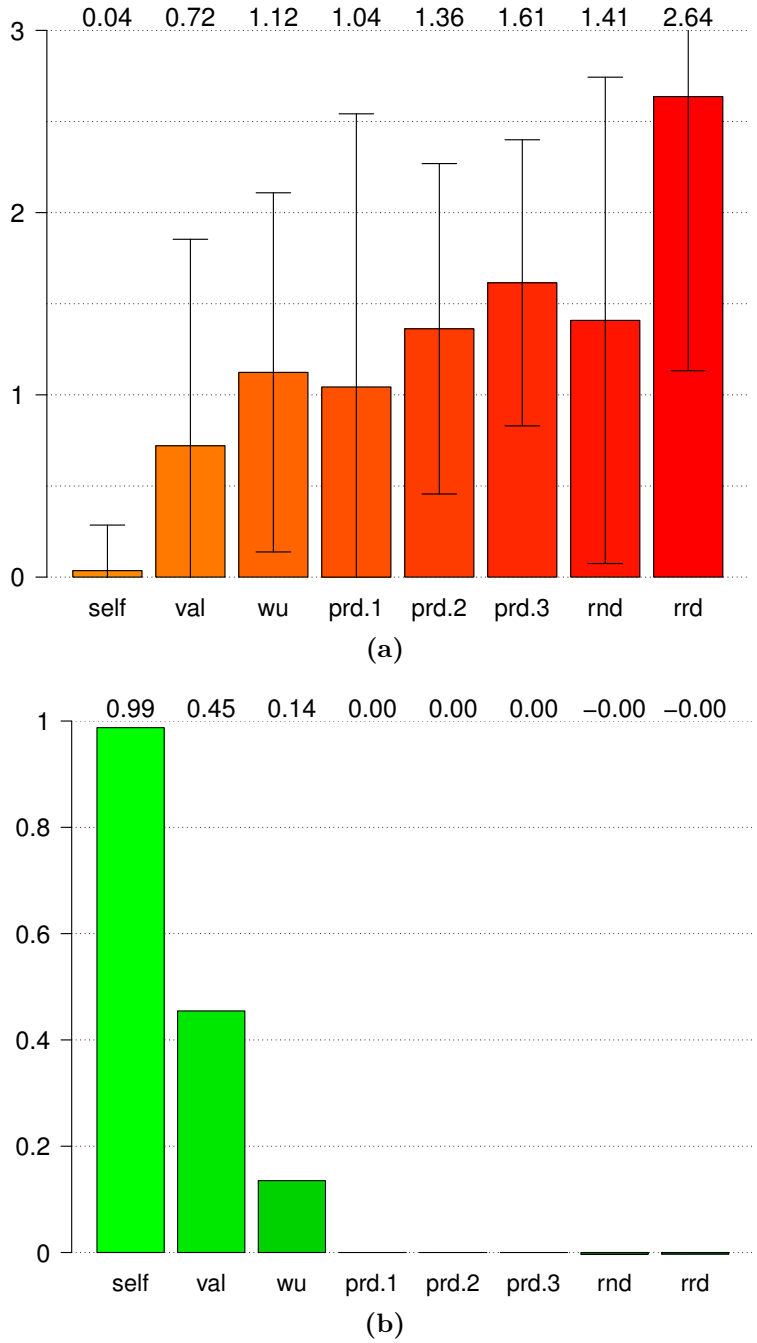


Figure 7.4: Recursion – **(a)** mean absolute error and **(b)** correlation

As with learning loop iteration counts, we have considered simple heuristics that select a value at random for the recursion frequency (thus classifying the function in one of the 9 classes) and have performed the same experiments. The results of random prediction based on the distribution of classes is shown in Figure 7.4 (label *rnd*). The mean absolute error is 1.41 with a standard deviation of 1.33, and the correlation is 0.00. Considering the Δk-error, less than 40% of the predictions are correct and about 60% have a maximal error of one class (see Figure 7.5). The experimental results of our approach are drastically better than the prediction that depends on the distribution of the recursion frequency, which is an evidence that the machine learned predictor

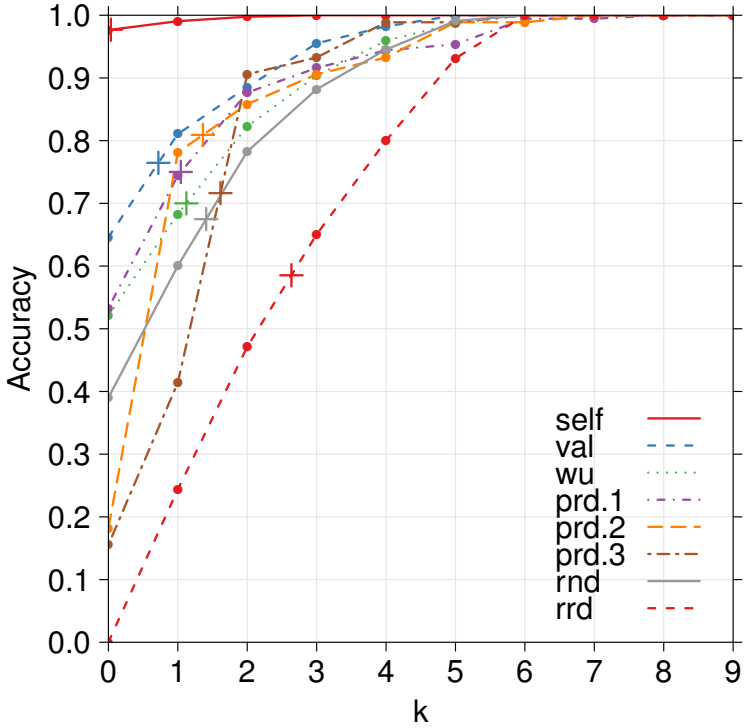


Figure 7.5: Recursion – Δk-accuracy

has not found alone the most frequent classes. The pure random predictor yields the worst result (see Figure 7.4, label *rrd*). The mean absolute error is 2.64 with a standard deviation of 1.50, and the correlation is 0.00. Examining the Δk-accuracy, about three quarters of the predictions have an error above one class (see Figure 7.5).

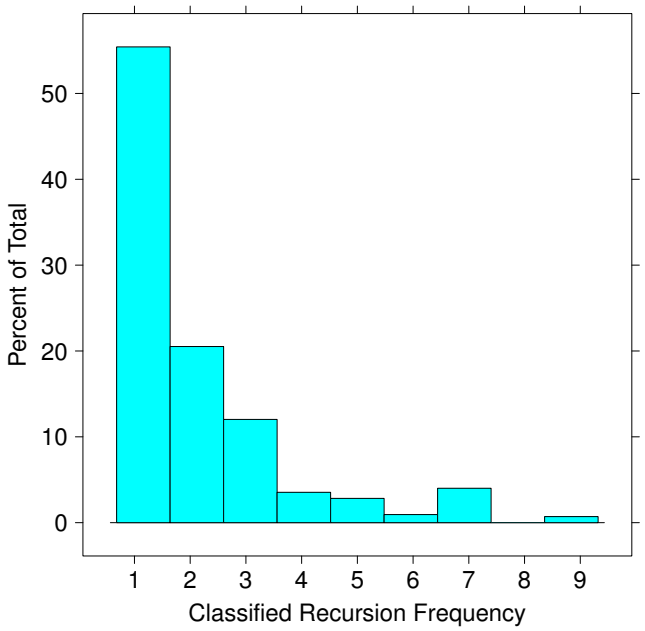


Figure 7.6: Distribution of classified recursion frequency

In summary, all experimental results shown in this subsection demonstrate that other considered heuristics for predicting the recursion frequency perform inferior to our approach. This demonstrates that our learning algorithm can successfully identify the relationship between features and classes for a given program.

7.2.3 Prediction of Execution Time

In the following, we show the precision of the machine learned predictors for execution times. We compare different basic learning algorithms for regression modeling, namely *decision trees*, *k-Nearest Neighbor* (*k-NN*), *Ordinary Least Squares Estimation* (*OLS*) for linear regression, the *Iterated Reweighted Least Squares* (*IWLS*) method for robust linear modeling, and *Support Vector Machine* (*SVM*). We furthermore have adapted and implemented the *Predicting Query Run-time 2* (*PQR2*) technique of Matsunaga and Fortes [MF10] and we compare its accuracy against the other basic learning algorithms.

We have analyzed the functions that are contained within the 190 programs of our benchmark collection and we have measured their execution times in nanoseconds. In total, we have observed 23037 execution times that range from 45 nanoseconds to more than 5 hours, with an average of 8.4 seconds. For learning, we have considered 13 features representing occurrences of categorized *Intermediate Representation* (*IR*) constructs and we have proposed two different weighting schemes (in Subsection 4.2.1). The first scheme, which we call *dynamic feature weighting*, multiplies each occurrence with its execution frequency, which we derive from interprocedural path profile information. The second scheme, which we call *static feature weighting*, multiplies each occurrence with the static execution frequency prediction that we compute according to the algorithm of Wu and Larus [WL94]. The first scheme is not applicable in a pure static setting, since information from path profiling is required at compile time. We use it, however, to assess whether our chosen instruction categorization is basically suitable for learning execution times. Since we aim at improving the mapping when solely static information is available, the evaluation with static feature weighting refers to the main experiment, showing the accuracy of the predictor for execution times of MaCAPA[6].

The different ML techniques define model parameters with which the learning algorithms can be configured. The k for the k-NN technique must be given and the maximal iteration count of the IWLS technique can be set. For SVMs, the ε of the ε-*insensitive* loss function and the constant C must be defined by the user (see Subsection 2.3.2). Instead of choosing a fixed value for each parameter, an automatic *grid search* within a given interval can be used to find the best value. Our adaption of PQR2 can be configured with several model parameters, such as the number of the largest time gaps *ngap* as the points at which to split the time range into two or the minimal accuracy *minAcc* and

[6] Of course, a user of our framework can decide to use dynamic feature weighting to obtain more precise results (as shown in our experiments using the first scheme), though having the cost to perform path profiling before.

Table 7.3: Configuration of ML techniques

Name	Label	Model parameters
Linear regression (OLS)	lm	-
Decision tree	dTree	-
Robust linear regression	rlm	maximal iteration count 200
k-Nearest Neighbor	knn	grid search for $k \in [1, 2, \ldots, 10]$
Support Vector Machine	svm	grid search for $C \in [-15, -5, 3, 15]$ and $\varepsilon \in 2^{[-15,-13,-11,\ldots,3]}$
Predicting Query Run-time 2	*pqr*	$ngap = 4$, $nskip = 25$, $minAcc = 80$, $minExamples = 600$, $minInterval = 100$

minimal interval size *minInterval* as threshold for further partitioning the current range (see Subsection 6.3.2). We show the setting of the model parameters for our experiments in Table 7.3[7].

In the following, we first evaluate the accuracy of the ML techniques with dynamic feature weighting. After that, we present our main experiment, which evaluates the precision of the predictor for execution times of MaCAPA. All experimental results are derived from *leave-one-program-out cross-validation*. As precision measures, we use the mean absolute error and the correlation between predictions and correct outcome, and, additionally, the *median absolute error*. For assessing average errors, the median is usually preferred to other measures when some outliers (i. e., some high errors) skew the mean.

Dynamic Feature Weighting

In this subsection, we present our experiments to evaluate the ML techniques where the features are weighted with run-time information obtained from path profiling. In Figure 7.7, we show the mean and median absolute errors of the considered ML techniques. We compare the techniques to a *naïve* approach where the predictor simply chooses a fixed value as result, namely the average of observed times in the training data (which is about 8.4 seconds). The caused errors by the naïve approach can be seen as an upper bound for acceptable errors of ML techniques. If an ML technique caused higher errors than the naïve approach, the *Training Phase* of MaCAPA to establish ML based predictors for execution times would be rarely worth the effort. The decision tree learning technique is labeled with *dTree*, k-NN with *knn*, the IWLS method with *rlm*, OLS with *lm*, SVM with *svm*, and our adaption of the PQR2 algorithm with

[7]The ranges within we search the best ε and the best C for SVM and the best k for k-NN are the default ranges of the *rminer* package from the *Comprehensive R Archive Network*.

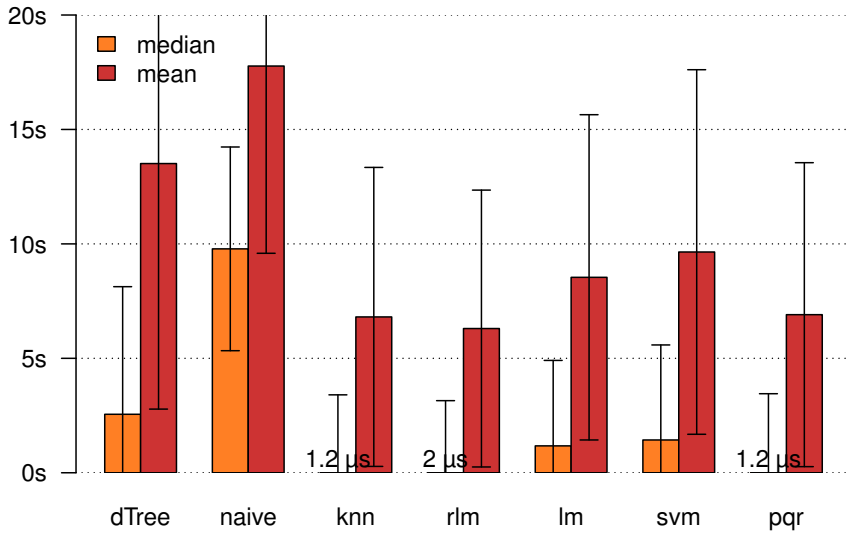


Figure 7.7: Time – dynamic weighting: mean and median absolute errors

pqr. The label *naïve* denotes the naïve prediction. As can be seen, the mean errors are skewed compared to the median errors due to the great range of errors. Hence, the median error is more meaningful to assess the quality for learning execution times based on our collected observations. All considered ML techniques yield lower mean and median absolute errors than the naïve approach. That is, they make a more intelligent estimation than simply predicting by the average and hence can be considered to be useful in principle. The lowest median errors are caused by the k-NN technique and by our adaption of PQR2 with 1.2 microseconds, followed by the IWLS technique with 2 microseconds (the standard deviations are indicated by bars). These three approaches also yield the lowest mean errors. The IWLS technique causes the lowest mean error with 6.3 seconds, followed by k-NN with 6.8 seconds and our adaption of PQR2 with 6.9 seconds. This shows that they do not have as many high prediction errors as the other approaches. In the following, we discuss the errors in more detail.

In Table 7.4, we show observed execution times and prediction errors partitioned by their magnitude. Each element in the table represents the number of observations that fall below the threshold denoted by the column heading and that have a higher value than the threshold of the column to the left (i. e., the numbers are not cumulative). Hence, the values in each row sum up to 23037 observations. The first row denotes actual observed execution times. As can be seen, more than half of the actual execution times are observed in the range between one microsecond and one millisecond and nearly one third of the values are below one microsecond. In contrast to these short times, about five percent of the executions take longer than one second. In particular, 349 of the functions take more than one minute to execute (with up to more than 5 hours). In the other rows, we present the number of absolute prediction errors of the ML techniques compared to the naïve approach. The magnitudes of prediction errors for decision trees, OLS, and SVM are comparable to the

Table 7.4: Time – dynamic weighting: actual times and absolute errors

	$\leq 1\mu s$	$\leq 1ms$	$\leq 1s$	$\leq 1min$	other
actual time	6610	13002	2266	810	349
dTree	0	2	106	22621	308
naïve	0	0	49	22694	294
k-NN	11171	8393	2343	835	295
rlm	3354	16569	2031	760	323
lm	0	0	1078	21591	368
SVM	0	1	309	22428	299
adapted PQR2	10990	8884	2117	748	298

naïve prediction errors, though they yield lower mean and median errors than the naïve approach. Solely the k-NN technique and our adaption of PQR2 are able to predict execution times with nearly half of the prediction errors below one microsecond. The IWLS method for robust linear modeling yields errors mostly in the range between one microsecond and one millisecond. This is reflected by the low median errors of the latter techniques. The number of errors that are higher than one minute are comparable across all predictions. However, they differ in their magnitude, which skews the mean errors of the predictions differently. In summary, this table shows in more detail that the k-NN technique and our adaption of PQR2 yield the most precise predictions. However, the table does not show whether low prediction errors correspond to short actual observed execution times, i. e., whether the values in each column stem from the same (or similar) functions, which we analyze next.

In the following, we discuss the *location of errors* for different ML techniques and for the naïve approach to assess the precision relative to the actual execution time. To that end, we have sorted the 23037 actual observations by time in ascending order and we have sorted the prediction errors accordingly. For a fine-grained view on the location of errors, we visualize the actual times and the errors color coded in Figure 7.8. Each box in the figure is filled column-wise with the sorted observations (i. e., the boxes have no x- nor y-dimension). The actual execution time of each observation is shown in Figure 7.8a. The other boxes visualize the absolute errors of leave-one-program-out cross-validation with different ML techniques. The predictions with *decision trees* is shown in Figure 7.8b, the naïve prediction in Figure 7.8c, k-NN in Figure 7.8d, robust linear modeling in Figure 7.8e, OLS in Figure 7.8f, SVM in Figure 7.8g, and our adaption of the PQR2 algorithm in Figure 7.8h. As the location of the errors demonstrate, k-NN and our adaption of PQR yield the smallest errors for the functions with the shortest execution times and, dual to that, the highest errors for long running functions. The IWLS method for robust linear modeling (which is more robust against outliers than OLS)

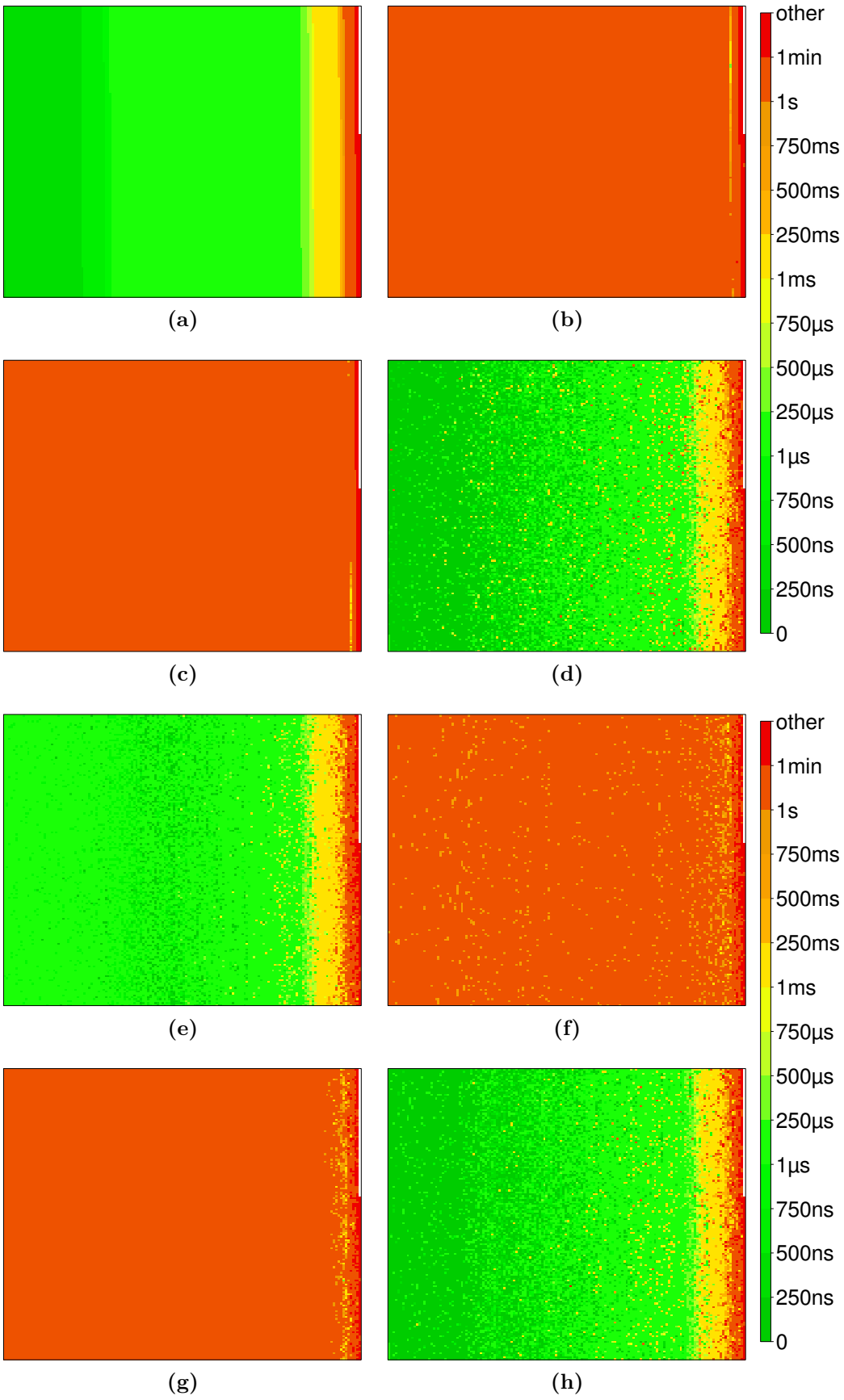


Figure 7.8: Time – dynamic weighting: **(a)** observed values vs. prediction errors of **(b)** decision trees, **(c)** naïve, **(d)** k-NN, **(e)** IWLS, **(f)** OLS, **(g)** SVM, **(h)** our adaption of PQR2

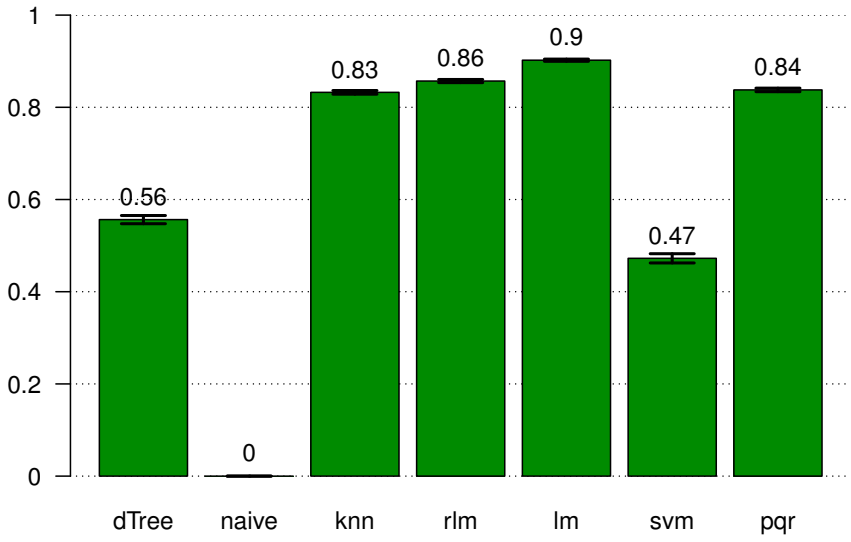


Figure 7.9: Time – dynamic weighting: correlation

yields the lowest errors near the center of sorted predictions, which indicates that the model was tailored for the most common case. Interestingly, OLS and SVM (and to a small amount the decision tree technique also) yield several lower errors for longer running functions. This reveals that they have learned some longer running functions better while neglecting the other observations and therefore perform poor on average. In conclusion, solely the predictions of k-NN, IWLS, and our adaption of PQR2 are comparatively precise relative to the actual execution time.

In addition to the mean and median errors, we consider the correlation between predictions and the correct outcome. In particular, we have determined *Pearson's product-moment correlation coefficient* [LRN88] as the measure of the degree of linear dependency between the predictions and the actual observed execution times. We show the correlations of the ML techniques with dynamic feature weighting in Figure 7.9. The bars denote the confidence interval for the correlations at the 95% confidence level. All considered ML approaches yield predictions which are strongly correlated with the correct outcome and, of course, the correlation for the naïve predictor is zero. The highest correlation achieve the OLS and the IWLS techniques (which is unsurprising since both techniques directly learn a linear relationship). The third highest correlation achieves our adaption of PQR, followed by k-NN. The high correlations for all ML approaches demonstrate that a relationship can be learned based on our chosen categorization of IR constructs.

This subsection has shown that execution times can be predicted precisely (relative to the actual execution times) with a strong correlation to the correct outcome if information from path profiling is used to weight the features for learning. In the following subsection, we evaluate the predictors if solely static information is used to weight features.

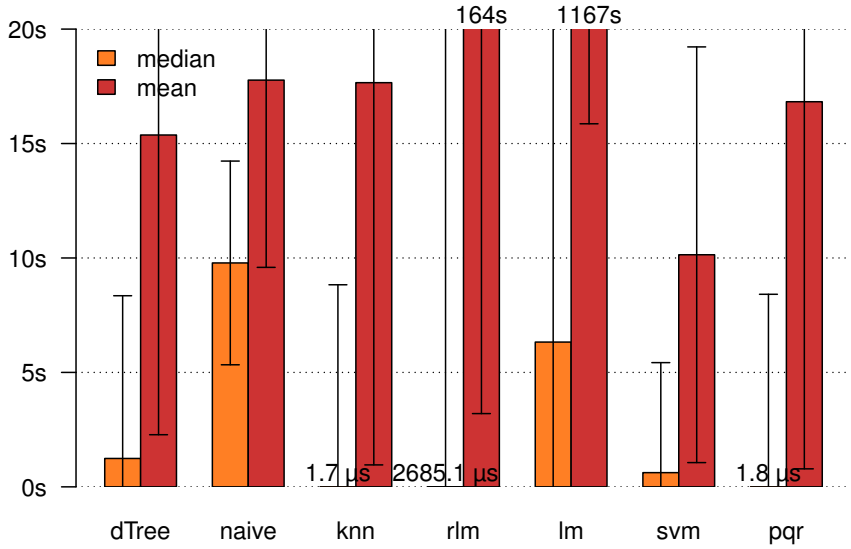


Figure 7.10: Time – static weighting: mean and median absolute errors

Static Feature Weighting

In this subsection, we present our main experiment, which shows the accuracy of the predictor for execution times of MaCAPA. The ML models we consider are learned with features that represent categorized IR constructs, weighted by their statically predicted execution frequency. As a consequence, the established predictors solely require statically weighted IR constructs as features to make a prediction for unseen data. Hence, they can be employed by our framework MaCAPA, which aims at improving the mapping without requiring run-time information for the current program at compile time. The Figure 7.10 shows the mean and median absolute errors for *leave-one-program-out cross-validation* with static feature weighting (the standard deviations are indicated by bars). Again, the regression techniques are compared with the naïve prediction by the average and are labeled as before. The SVM learning technique yields the lowest mean absolute error with about 10.1 seconds, followed by decision trees with 15.4 seconds, our adaption of the PQR2 algorithm with 16.8 seconds, and k-NN with 17.7 seconds. Solely these techniques cause on average lower mean prediction errors than the naïve approach. The mean errors of OLS and IWLS for linear regression drastically increase to 1167 seconds and 164 seconds, respectively.

In contrast, the median errors are smaller in orders of magnitude. Compared to dynamic feature weighting shown in the previous subsection, the median errors caused by the ML techniques do not increase dramatically. For decision trees and SVM, the median error in fact decreases, for the former from 2.6 seconds to 1.2 seconds and for the latter from 1.4 seconds to 0.6 seconds. Furthermore, all ML techniques yield lower median absolute errors than the naïve approach. The k-NN technique yields the lowest median error with 1.7 microseconds, followed by our adaption of the PQR2 algorithm with 1.8 microseconds and IWLS with about 2.7 milliseconds. As stated be-

fore, the median error represents the central tendency of errors better than the mean error if some high errors skew the mean. In summary, the mean and median errors reveal that four learning techniques can considered to be useful in principle since both their mean and median absolute errors are lower than those of the naïve approach:

1. **Our adaption of PQR2:** It causes (together with k-NN) the lowest median absolute error and the third best mean absolute error.
2. **k-NN:** It causes the lowest median error but only the fourth best mean error.
3. **SVM:** It causes the lowest mean error but only the fourth best median error.
4. **Decision trees:** It causes the second best mean error but only the fifth best median error.

In the following, we evaluate the absolute errors in more detail to decide which technique is best suited for establishing the execution time predictor of our framework MaCAPA.

In Table 7.5, we present the absolute errors partitioned by their magnitude. Again, an element in the table represents the number of observations that fall below the threshold denoted by the column heading and that have a higher value than the threshold of the column to the left. We again show the actual observed execution times (in the first row) and the naïve prediction errors (row *naïve*). The magnitude of errors for the decision tree and the OLS techniques are comparable to the naïve prediction. The SVM and the IWLS techniques mainly yield errors between one millisecond and one second, i. e., more precise predictions than the naïve approach. Since IWLS has a much higher mean absolute error than the naïve prediction (164 seconds compared to 17.8 seconds),

Table 7.5: Time – static weighting: actual times and absolute errors

	$\leq 1\mu s$	$\leq 1ms$	$\leq 1s$	$\leq 1min$	other
actual time	6610	13002	2266	810	349
dTree	0	0	387	22223	427
naïve	0	0	49	22694	294
k-NN	10529	7905	2789	1303	511
rlm	0	4558	16735	1328	416
lm	0	0	103	22394	540
SVM	0	0	21473	1123	441
adapted PQR2	10159	8396	2861	1213	408

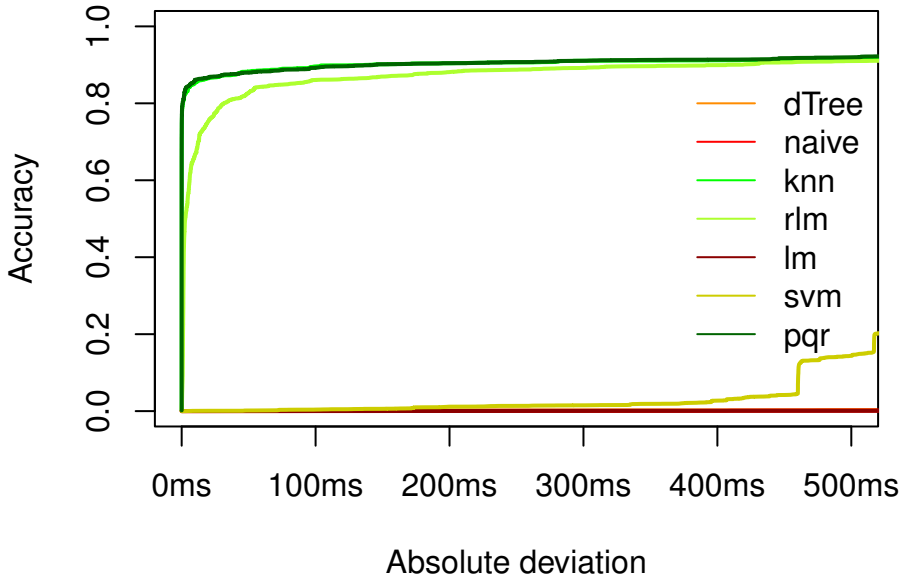


Figure 7.11: Time – static weighting: regression error characteristic curves

the magnitudes indicate that IWLS has huge outliers as prediction errors. As in the previous subsection, solely our adaption of the PQR2 algorithm and the k-NN learning technique are able to predict execution times of nearly half of the functions with an absolute error below one microsecond. Furthermore, our adaption of the PQR2 algorithm has the smallest number of errors above one minute and the greatest number of errors below one millisecond compared to all other ML techniques.

The Table 7.5 denotes fixed thresholds for the partitioning of errors and the values in the table are not cumulative. Defining thresholds in a continuous range and cumulating the errors that are below a threshold in the range leads to *regression error characteristic* (*REC*) curves [BB03]. REC curves plot the error tolerance on the x-axis versus the percentage of points predicted within the tolerance on the y-axis. Hence, the resulting curve estimates the cumulative distribution function of the error. Figure 7.11 shows a section of the REC curves of the absolute deviation for our investigated ML techniques. This demonstrates that our adapted PQR2 technique and the k-NN technique predict more than 85% of the observations with an absolute deviation less than 250 milliseconds. Remarkably, the IWLS technique for robust linear modeling is competitive with our adaption of PQR2 in the shown tolerance.

As in the previous subsection, we evaluate whether low prediction errors correspond to short actual observed execution times, i. e., whether the ML predictors are precise relative to the actual execution time. We show a fine-grained view on the *location of errors* for leave-one-out cross-validation with different ML techniques in Figure 7.12. Again, we depict the observed actual execution times color coded in Figure 7.12a, sorted in ascending order. For visualizing to which observation the prediction errors are related, we have sorted the errors accordingly. The predictions errors with the decision tree technique is shown in Figure 7.12b, the naïve prediction in Figure 7.12c, k-NN

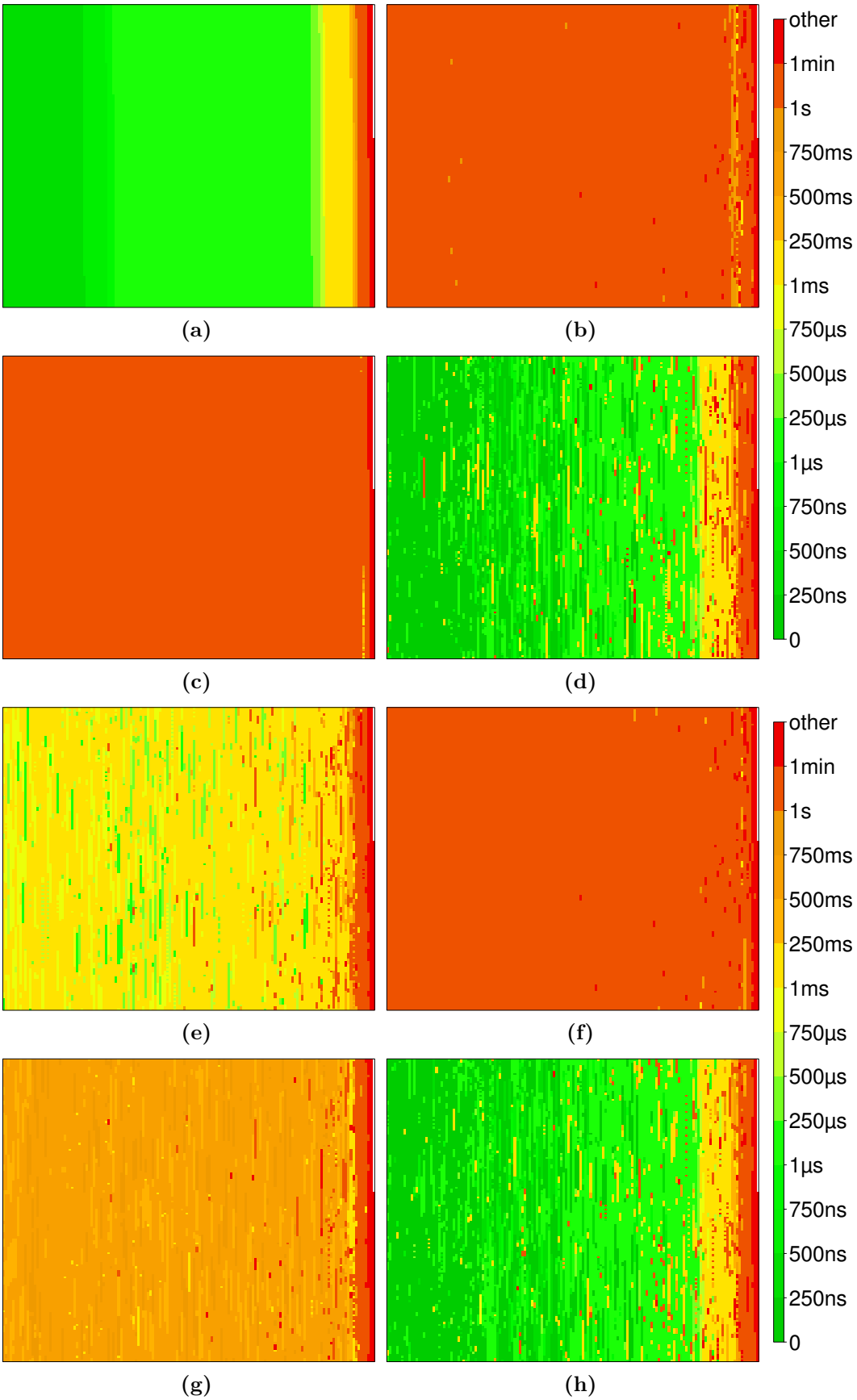


Figure 7.12: Time – static weighting: **(a)** observed values vs. prediction errors of **(b)** decision trees, **(c)** naïve, **(d)** k-NN, **(e)** IWLS, **(f)** OLS, **(g)** SVM, **(h)** our adaption of PQR2

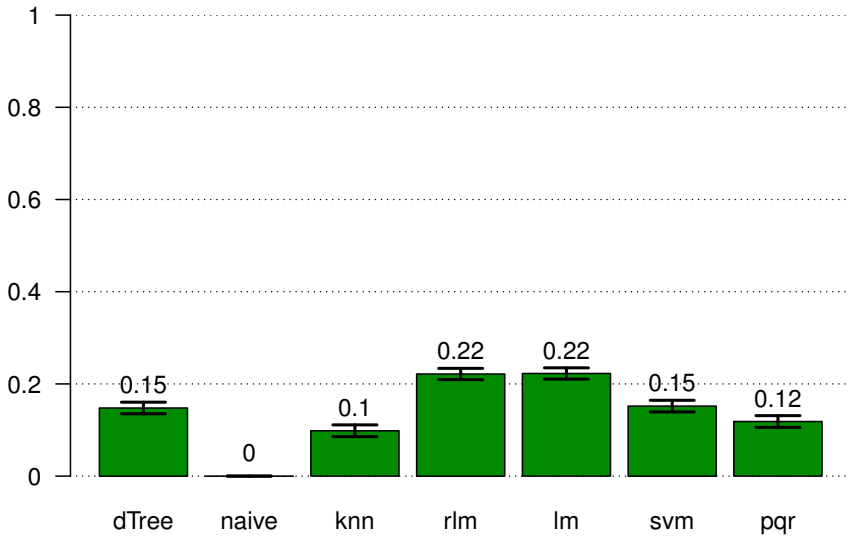


Figure 7.13: Time – static weighting: correlation

in Figure 7.12d, IWLS for robust linear modeling in Figure 7.12e, OLS in Figure 7.12f, SVM in Figure 7.12g, and our adaption of the PQR2 algorithm in Figure 7.12h. As can be seen, the two learning techniques that have a lower mean absolute error than our adaption of the PQR2 algorithm, *decision tree* and SVM, perform worse than our approach. Furthermore, our adaption of PQR2 mainly yields large prediction errors at functions with large execution times. The only technique that is competitive with our adapted PQR2 is the k-NN technique. However, k-NN yields higher errors at functions with shorter execution times more frequently than our adaption, as can be seen when comparing the frequencies of yellow and red colored errors to the left side of each box. This demonstrates that our predictions with PQR2 are most precise relative to the actual execution time.

Finally, we also have determined *Pearson's product-moment correlation coefficient* [LRN88] for the ML techniques with static feature weighting. Figure 7.13 depicts the correlations, where the bars denote their confidence interval at the 95% confidence level. The highest correlations achieve the OLS and IWLS techniques for linear regression with 0.22, followed by the decision tree and SVM techniques with 0.15. Our adaption of PQR2 achieves a correlation of 0.12. All correlations except the zero correlation of the naïve predictor are statistically significant at the probability level $p = 0.001$. In other words, we have 99.9% evidence that a relationship exists[8]. In conclusion, the execution time of functions can successfully be learned by all ML techniques (though with different precision as demonstrated before).

In this subsection, we have evaluated the accuracy of the predictors for execution times when solely static information is available at compile time to weight the features for learning the ML models. We have demonstrated that

[8]To determine whether the correlation indicates that a relationship was learned, *critical values* of Pearson's correlation coefficient give the thresholds for a given probability level (which is 0.02 at a probability level of 0.001 for our training data).

our adaption of the PQR2 technique outperforms all other ML techniques by yielding the most precise predictions (relative to the actual execution time). We also have shown that our adaption causes the lowest median absolute error (together with k-NN) and is able to predict nearly half of the functions with an absolute error less than one microsecond. Moreover, it predicts more than 85% of the functions with an absolute deviation less than 250 milliseconds. We therefore use our adaption of the PQR2 technique (with static feature weighting) as the predictor of execution times for our framework MaCAPA. In Section 7.3, we evaluate the program performance improvement that can be achieved with MaCAPA when our ML based predictors are used. In the following, we present our experiments with execution time predictors learned on different CPUs.

7.2.4 Prediction of the Best Performing PE

We have established ML predictors of execution times based on static feature weighting for different PEs to determine which PE will execute an application most efficiently. This enables us to rate a number of CPUs according to their computational performance. We have considered six different Intel® CPUs as PEs: Pentium D @3.40GHz, Core 2 Duo T7300 @2.00GHz, Core 2 Duo E6850 @3.00GHz, Core 2 Duo E7500 @2.93GHz, Xeon E5430 @2.66GHz, and Core i5-2400 @3.10GHz[9]. We have profiled the 190 programs of our benchmark collection on each PE. In Table 7.6, we show a summary of observed execution times. As can be seen, the Core i5-2400 CPU performs best on average. Interestingly, this is not the case for all functions. The row *Best PE* in this table shows the number of functions that are executed most efficiently on the corresponding PE. Based on the observed execution times on each CPU and the feature vector of each function, we have established six ML models with our adaption of the PQR2 algorithm. With these models, we obtain execution time predictions for each PE from a feature vector. Again, we have performed cross-validation, i. e., we have applied the learned predictors to unseen data for evaluation. In particular, we have taken about two thirds from the observations as training data to establish the models, namely 15537 functions. The resulting predictors are applied to the remaining one third of observations, namely to 7500 functions. The training data is taken from the first benchmark suites *BioBench*, *Bit Stream*, *cBench*,and the first 1077 benchmarks of *CSiBE* (which sum up to 15537 observations). The validation set for the predictors are the remaining 7500 functions. Then, we have determined the PE that yields the lowest execution time prediction. Using this scheme with the feature vector of the `main` function of an application hence predicts the best performing PE for this application.

Our experimental results again demonstrate the accuracy of our approach. For 3975 of the 7500 functions, we have correctly predicted the PE that ex-

[9]The experiments in the previous subsection were performed on the Core 2 Duo E6850 CPU because it is the host of MaCAPA.

Table 7.6: Execution times on different PEs and results for predicting the best PE

	Pentium	T7300	E6850	E7500	E5430	i5-2400
Min.	$66ns$	$63ns$	$45ns$	$44ns$	$48ns$	$27ns$
1st Qu.	$836ns$	$1816ns$	$589ns$	$582ns$	$545ns$	$470ns$
Median	$5.08\mu s$	$10.50\mu s$	$3.57\mu s$	$3.12\mu s$	$3.05\mu s$	$2.33\mu s$
Mean	$21.52s$	$15.65s$	$8.35s$	$2.72s$	$2.89s$	$2.26s$
3rd Qu.	$46.21\mu s$	$68.86\mu s$	$37.19\mu s$	$25.12\mu s$	$23.19\mu s$	$19.00\mu s$
Max.	$53714s$	$20084s$	$20780s$	$3327s$	$3576s$	$2745s$
Best PE	1	1	230	95	194	6979
#Pred.	280	84	1065	1278	576	4217
Exact	0	0	29	17	33	3896

ecutes a function most efficiently. Note that our prediction is not the trivial one, i. e., we have not always predicted the Core i5-2400 CPU as the best. In row *#Pred.* of Table 7.6, we show the number of predictions that rate the corresponding PE as the best. The row *Exact* presents the number of these predictions that are correct (which sum up to 3975). That is, we are able to exactly predict the PE that executes a function most efficiently for 53% of the functions. For 1273 of the functions (i. e., 17%), we have predicted the second best performing PE to be the best, for 739 (i. e., 9.9%) the third best performing PE as the best, and for 975 (i. e., 13%) the fourth best as the PE that executes a function most efficiently. In only 457 cases (i. e., 6%), we have predicted the second worst PE to be the best and in only 81 cases (i. e., 1.1%) the worst PE to be the best. Hence, our approach mainly predicts the better performing PEs and precisely excludes the worst PEs.

Given a heterogeneous architecture, a run-time scheduler does not know in general on which PE an application will execute most efficiently and thus can allocate it everywhere. Hence, we can assess the resulting benefit when an application is allocated to the PE that our predictor rates as the best. The mean benefit of an application is the difference between the execution time of the `main` function on our predicted PE and the mean of its execution times on all PEs. Similarly, the maximal benefit of our approach is the difference to the maximum of its execution times on all PEs. To determine this, we have trained the models with all but the `main` functions and have applied the learned predictors to the `main` functions. The sum of the mean execution times of the `main` functions on all PEs are $8.7488 \cdot 10^4$ seconds (i. e., about 24.3 hours), and the sum of the maximal execution times are $2.2458 \cdot 10^5$ seconds (i. e., about 62.4 hours). The sum of the execution times of the `main` functions on our predicted PEs are $5.2898 \cdot 10^4$ seconds, which is about 14.7 hours. Hence, using our approach yields a mean execution time reduction of 39.5% since the `main`

functions take only 60.5% to execute on our predicted PEs compared to the mean execution times. Considering the worst case, our ML based predictor can yield a maximal execution time reduction of 76.4% because the `main` functions take only 23.6% to execute on our predicted PEs compared to the maximal execution times. In conclusion, our approach is best suited to predict execution times based on static code features and, based on this, to predict the PE that executes an application most efficiently.

In this section, we have discussed the precision of our ML based predictors for unknown run-time behavior. We have demonstrated that our ML approaches for learning loop iteration counts, recursion frequencies, and execution times successfully identify the relationship between static code features and regarded run-time behavior with a high accuracy. In the following section, we investigate the performance gain that our instantiated framework MaCAPA achieves by using our ML based predictors.

7.3 Program Performance Evaluation

The previous section has shown the precision of our ML techniques, which our instantiated framework MaCAPA employs for learning unknown run-time behavior. We have performed experiments to assess the actual performance improvement that is achieved if MaCAPA is used to automatically map MPI processes to a heterogeneous network of processors. In this section, we first describe the experimental setup and then present the results.

7.3.1 Experimental Setup

We have performed experiments with the 34 MPI benchmarks that we have introduced in Table 7.2. We have compared the performance when the processes are mapped with MaCAPA against the performance for the default MPI allocation strategy, i. e., when a user does not specify the allocation of MPI processes to the PEs. Hence, we have measured the required execution times for each program if MaCAPA determines the allocation and if the MPI runtime daemon performs a *Round-Robin* allocation to the available PEs (which is the default strategy).

As target architecture, we have used a network of CPUs, which are fully connected with each other via Ethernet. We have used the same CPU models as for learning the best performing PE (i. e., the CPUs in Table 7.6). With that, our learned predictor is able to rate the models according to their computational performance. We have established a processor network with 16 nodes (i. e., servers) as shown in Table 7.7. The table lists for each CPU model how frequently it is being used (row *#Used CPUs*, which sum up to 17) and how many cores it has (row *#Cores*). Since the model XEON E5430 with four cores is integrated twice in a dual socket motherboard of a single server, we have 16 nodes in the network. The row *Node index* denotes the indexes

Table 7.7: CPUs of the processor network

	Pentium	T7300	E6850	E7500	E5430	i5-2400	$\sum$
#Used CPUs	1	1	2	2	2 (in 1 node)	9	17
Node index	N_1	N_2	$N_{3,4}$	$N_{5,6}$	N_7	N_{8-16}	16
#Cores	2	2	2	2	4	2	
#Res. PEs	2	2	4	4	8	18	38

of the servers. As a result, we have 38 different PEs (i. e., cores) to which an MPI process can be allocated (row *#Res. PEs*).

To determine the mapping, the heuristic allocation algorithm of our framework MaCAPA considers not only the computational costs of processes but also the communication costs between processes. Hence, the bandwidths between the nodes in the processor network are relevant for the decision. The bandwidths are automatically obtained during the *Analysis Phase* of MaCAPA via the asymptotic throughput of message passing with different amounts, averaged over a certain number of iterations. The measured bandwidths are shown in Table 7.8. Each entry in the table represents the bandwidth for message passing from the node denoted in the first column to the node denoted by the header of the column of the element. The highest bandwidths arise, of course, when the source node is identical to the target node. For the *i5-2400* CPUs (Nodes N_8 to N_{16}), we measured nearly 930MB/s in that case, for the *Pentium D*, the *Core 2 Duo T7300*, and the *XEON E5430* (Nodes N_1, N_2, and N_7, respectively) between about 160MB/s and 260MB/s, and for the other, the *Core 2 Duo E6850* and *E7500* (Nodes N_3 to N_6), around 520MB/s. Remarkable are the very low bandwidths to and from the XEON dual-quad-core server (Node N_7). The bandwidths are (with about 0.33MB/s on average) less than 1.5% of the average bandwidths if the source is not identical to the target node (around 23MB/s). The low bandwidths to and from Node N_7 are caused by the fact that the server is in a different subnet of the Ethernet. Moreover, the nodes of the network form two clusters, which arise from the physical locations of the servers. Nodes N_1 to N_6 and N_8 constitute one cluster and Nodes N_9 to N_{16} another cluster (from this point of view, Node N_7 constitutes its own cluster). The bandwidths in Table 7.8 also reflect this clustering, though to a smaller degree than the very low bandwidths to Node N_7. Within each cluster, the bandwidths are mainly between 22.5MB/s and 25MB/s. Between the clusters, the bandwidths decrease slightly to the range mainly between 21MB/s and 22.5MB/s.

To start a program, the number of processes to execute it in parallel must be provided. We have performed experiments with two different values for this number. The observed bandwidths induce two allocation schemes when solely

Table 7.8: Bandwidths between PEs in MB/s

from\to	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8
N_1	161.36	22.88	24.33	24.48	25.49	26.88	0.30	22.19
N_2	23.77	396.04	23.23	23.26	24.58	25.13	0.34	21.75
N_3	25.43	22.85	540.52	26.01	27.81	28.38	0.34	23.03
N_4	24.68	22.83	25.51	553.52	27.20	27.63	0.33	22.92
N_5	27.90	27.39	27.87	27.70	511.85	30.25	0.33	23.77
N_6	28.12	27.41	27.86	27.38	30.19	491.71	0.33	23.21
N_7	0.34	0.34	0.34	0.33	0.30	0.33	262.07	0.34
N_8	23.31	22.86	22.60	23.07	23.11	23.77	0.34	938.34
N_9	22.39	22.85	22.00	22.01	22.31	22.06	0.34	19.90
N_{10}	22.53	23.01	21.83	22.01	21.98	21.95	0.34	20.00
N_{11}	22.52	23.10	21.79	22.12	21.96	21.97	0.34	20.05
N_{12}	23.20	22.90	21.96	22.24	22.34	21.98	0.34	19.98
N_{13}	23.15	22.90	21.94	22.14	22.32	21.90	0.34	19.97
N_{14}	23.28	22.79	21.93	21.93	22.22	21.93	0.34	20.28
N_{15}	23.29	22.80	21.88	21.94	22.31	21.87	0.34	20.14
N_{16}	22.39	22.85	22.00	22.01	22.31	22.06	0.34	19.90

from\to	N_9	N_{10}	N_{11}	N_{12}	N_{13}	N_{14}	N_{15}	N_{16}
N_1	22.10	22.45	22.45	21.93	21.93	22.03	22.03	22.10
N_2	21.50	21.50	21.50	22.33	22.33	22.91	22.91	21.50
N_3	22.06	21.99	21.99	21.68	21.68	21.71	21.71	22.06
N_4	22.22	22.14	22.14	22.10	22.10	22.11	22.11	22.22
N_5	22.18	22.35	22.35	22.31	22.31	22.34	22.34	22.18
N_6	22.20	22.16	22.16	22.30	22.30	22.10	22.10	22.20
N_7	0.34	0.34	0.34	0.34	0.34	0.34	0.34	0.34
N_8	20.20	19.78	19.78	19.90	19.90	19.86	19.86	20.20
N_9	931.97	23.85	24.08	24.08	23.80	23.80	24.05	24.05
N_{10}	24.08	927.29	23.78	24.11	24.11	24.10	24.10	24.08
N_{11}	24.08	23.78	927.29	24.11	24.11	24.10	24.10	24.08
N_{12}	23.85	24.03	24.03	924.92	23.82	23.85	23.85	23.85
N_{13}	23.85	24.03	24.03	23.82	924.92	23.85	23.85	23.85
N_{14}	23.78	23.78	23.78	23.82	23.82	925.62	24.08	24.05
N_{15}	23.78	23.78	23.78	23.82	23.82	24.08	925.62	23.78
N_{16}	23.85	24.08	24.08	23.80	23.80	24.05	24.05	931.97

the communication overhead is considered, depending on the number of processes to execute the MPI program. If 8 processes execute the program (and all processes communicate with each other), the processes are best launched at the dual-quad-core server (Node N_7) since the whole communication can bypass the slow Ethernet. If 16 processes are started to execute the program, the cluster of Nodes N_9 to N_{16} is the best target for the allocation (since the other cluster contains only 14 PEs such that another allocation scheme would cross cluster boundaries). Most important, Node N_7 should never be used for the allocation when more than 8 processes execute the MPI program (due to the very low bandwidths to and from N_7). We have performed the corresponding experiments, namely to execute the program with 8 and with 16 processes. During all experiments, no CPU in the processor network was stressed with other workload.

For each experiment, we compare the mapping derived by MaCAPA against the default MPI allocation strategy. The available PEs of the processor network must be provided to the MPI run-time environment in a *hostfile*[10]. The default MPI allocation strategy then schedules processes to run on each successive core on one server. When all those cores are filled, scheduling begins on the next server in the hostfile. As a consequence, the ordering of the servers within the hostfile directly influences the default MPI allocation and hence impacts the resulting default MPI performance (against which we compare the results of our approach). To overcome this, we have shuffled the hostfile at random[11] five times and have executed the benchmarks with each hostfile. We average the required execution times over these runs and compare it with the resulting execution time when MaCAPA automatically determines the mapping. For each setting, the MPI benchmarks are executed three times. For example, we have used the *test*, *train*, and *ref* data sets of the *SPEC* MPI2007 benchmarks shipped with the suite. Several other benchmarks do not require input data nor have configuration parameters. In this case, the three reported execution times stem from identical invocations of the benchmarks. In the following, we evaluate the experimental results.

7.3.2 Results

In this subsection, we evaluate the performance gain that our instantiated framework MaCAPA achieves. We have performed experiments with 34 benchmarks to be mapped to 16 nodes in a heterogeneous network of CPUs. We start with the experiment where 8 processes execute an MPI program. Afterwards, we evaluate the performance if 16 processes are launched to execute an MPI program.

Execution with 8 Processes

In Figure 7.14, we show the performance improvement of each benchmark that our framework MaCAPA achieves if 8 processes execute the MPI program. The rightmost bar denotes the average improvement across all bench-

[10] Our approach also automatically determines the available PEs from this hostfile.

[11] Technically, we have used the UNIX command `shuf` that shuffles the file by outputting a random permutation of its lines. Each output permutation is equally likely.

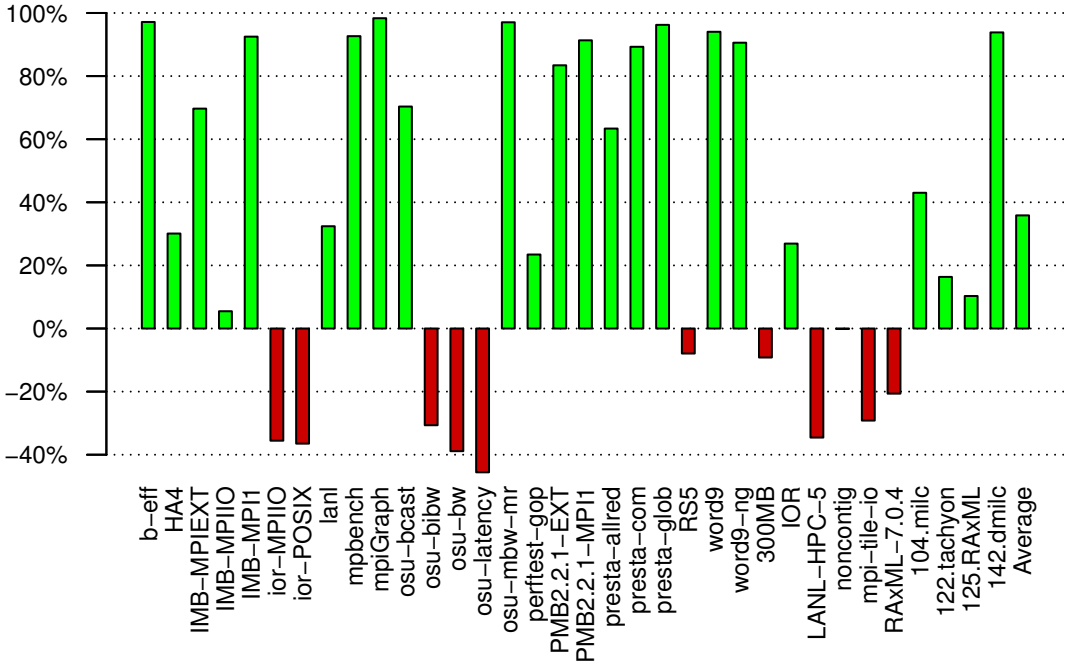


Figure 7.14: Performance – 8 processes: improvement gained with MaCAPA

marks. We see that for most of the programs, the run-time performance is significantly improved. For 15 out of 34 benchmarks, we have observed a huge performance improvement by more than 60% and up to 98% compared to the default MPI allocation strategy. For 5 benchmarks, MaCAPA achieves a notable improvement by more than 20%, and for 3 benchmarks a smaller improvement between 5% and 16%. Our mapping approach has a negligible effect for 1 benchmark. For 10 benchmarks, a performance degradation is ob-

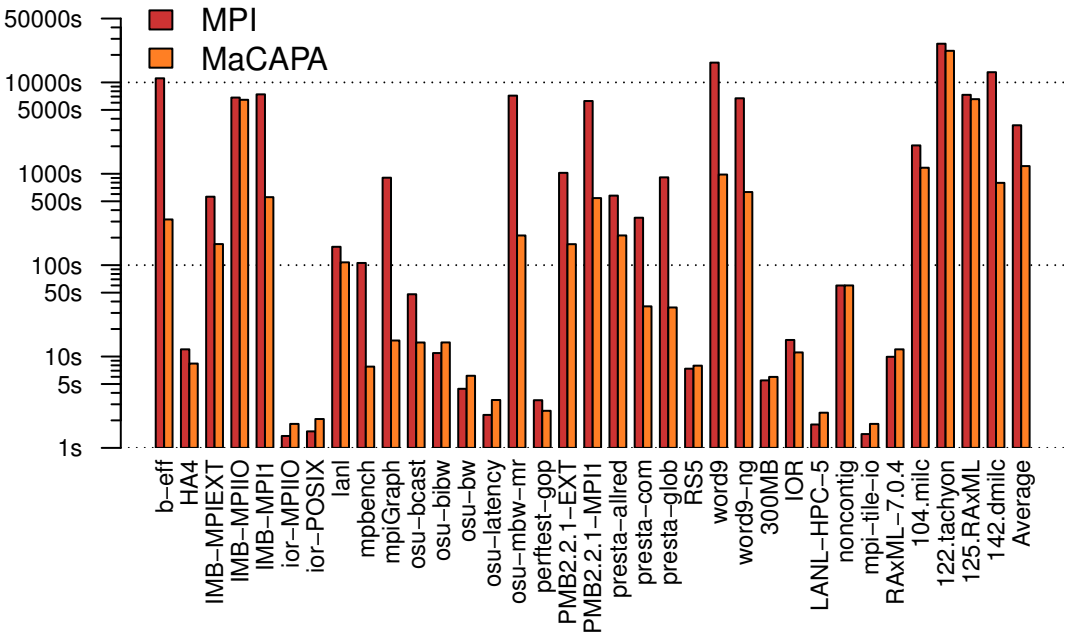


Figure 7.15: Performance – 8 processes: comparison between MPI and MaCAPA

served. The average performance improvement that we achieve with MaCAPA is 35.9%.

In Figure 7.15, we show the execution times in seconds. We compare the average execution time of each benchmark if it is mapped with the default MPI strategy against the time if it is mapped with MaCAPA (note the logarithmic scale of the y-axis). As can be seen, the benchmarks where a performance degradation is observed are invariably those with the shortest execution times. In contrast, the execution time is significantly reduced especially for long running benchmarks. The greatest execution time reduction by 15483 seconds is achieved with the benchmark *word9*, which means a shorter execution time of about 4.3 hours. On average (see the rightmost bars), MaCAPA reduces the execution time by 2182 seconds, i. e., by more than 36 minutes.

Table 7.9 shows the detailed results for the three runs of each benchmark with different input data. The table lists the execution times in seconds for the runs denoted by the column header. The 10 benchmarks where we have observed a performance degradation mainly have execution times below 5 seconds across all invocations. As can be seen, the increased execution times caused by our framework are in these cases below 2 seconds (except the *large data* run of benchmark number 14 with 2.31 seconds difference), with an average of 1.1 seconds. Hence, the performance degradation is low in all cases, compared to the huge execution time reduction that is gained with MaCAPA. Furthermore, we argue that a small difference between execution times does not necessarily indicate a worse allocation. It is shown by the fact that even for benchmarks without configuration parameters (hence having identical invocations), the execution times vary for the different runs with the MaCAPA allocation (which is, of course, identical for the three runs). This is the case for the benchmarks 1–18, 25, 27, and 29. Hence, small differences between execution times can also be caused by, e. g., network traffic on the Ethernet, instead of being caused by worse allocations.

The second greatest execution time reduction is observed at the first benchmark *Cbench/b-eff* with 10777 seconds on average. It is the program with which the effective bandwidths between the PEs are determined during the *Analysis Phase* of MaCAPA. The benchmark performs massive message passing between the executing processes. Considering the varying bandwidths between the nodes of the processor network, the performance of this benchmark is hence dominated by the communication cost. Our approach automatically reduces this cost by determining an allocation that bypasses low bandwidths for all three runs. Remarkably, the allocation of the default MPI strategy yields a much lower execution time for the *medium data* run than for the other runs, though all three runs are identical invocations of the program (because the program does not have a configuration parameter). This shows that the MPI strategy, without having predictions for communication costs, finds a good allocation at random. The same holds for the benchmarks *Cbench/IMB-MPI1* (No. 5, the *large data* run compared to the others), *Cbench/mpbench* (No. 9, the *small data* run), *Cbench/osu-mbw-mr* (No. 15, the *medium data* run), and *Cbench/PMB2.2.1-MPI1* (No. 18, the *small data* run).

Table 7.9: Performance – 8 processes: detailed execution times in seconds

No.	Benchmark	Small data		Medium data		Large data	
		MPI	MaCAPA	MPI	MaCAPA	MPI	MaCAPA
1	Cbench/b-eff	16641.78	325.24	387.54	306.91	16252.40	317.32
2	Cbench/HA4	9.49	8.76	10.05	8.69	16.41	7.68
3	Cbench/IMB-MPIEXT	273.38	171.30	744.30	169.85	665.63	169.09
4	Cbench/IMB-MPIIO	6715.80	6411.00	6711.00	6490.00	7046.00	6452.00
5	Cbench/IMB-MPI1	6710.75	552.59	15112.71	556.37	380.19	555.57
6	Cbench/ior-MPIIO	1.03	2.18	1.48	1.67	1.54	1.64
7	Cbench/ior-POSIX	1.43	1.93	1.17	2.64	1.95	1.64
8	Cbench/lanl	145.97	108.35	181.75	106.79	148.67	106.86
9	Cbench/mpbench	7.46	6.76	151.96	8.81	157.48	7.70
10	Cbench/mpiGraph	416.17	15.21	1488.07	14.79	812.47	14.93
11	Cbench/osu-bcast	10.95	15.05	71.66	13.67	61.44	13.99
12	Cbench/osu-bibw	9.68	13.61	7.22	14.89	15.85	14.29
13	Cbench/osu-bw	5.81	5.96	3.29	5.83	4.2	6.68
14	Cbench/osu-latency	2.57	3.23	2.22	2.39	2.10	4.41
15	Cbench/osu-mbw-mr	10645.48	211.36	171.01	211.16	10685.93	211.14
16	Cbench/perftest-gop	2.70	3.69	2.93	2.14	4.35	1.81
17	Cbench/PMB2.2.1-EXT	905.02	172.37	1122.19	170.00	1043.54	166.75
18	Cbench/PMB2.2.1-MPI1	533.41	542.15	6762.81	541.72	11471.98	542.37
19	Cbench/presta-allred	513.92	211.56	752.52	211.59	464.38	210.89
20	Cbench/presta-com	678.14	10.10	229.63	9.83	83.59	86.31
21	Cbench/presta-glob	451.97	11.00	232.73	10.58	2057.45	81.73
22	Cbench/RS5	4.35	5.09	6.25	6.68	11.48	12.06
23	Cbench/word9	19621.03	981.12	10483.44	981.60	19288.12	979.77
24	Cbench/word9-ng	991.16	630.52	14702.62	631.42	4412.60	632.10
25	Cbench/300MB	5.94	9.85	4.44	3.25	6.06	4.85
26	IOR	2.81	1.67	13.75	8.23	28.96	23.37
27	LANL-HPC-5	1.48	2.09	1.87	1.89	2.06	3.30
28	PIO/noncontig	2.86	2.48	20.30	16.91	156.46	160.50
29	PIO/mpi-tile-io	1.54	2.16	1.13	1.69	1.58	1.64
30	RAxML-7.0.4	16.66	22.25	5.97	6.88	7.17	6.82
31	MPI2007/104.milc	222.19	29.75	44.85	46.61	5864.16	3417.33
32	MPI2007/122.tachyon	2.39	2.23	23557.00	23892.00	55955.40	42621.00
33	MPI2007/125.RAxML	4224.09	3945.00	10078.00	7865.00	7630.8	7865.00
34	MPI2007/142.dmilc	412.63	39.26	581.95	36.45	37806.5	2311.75

In contrast, our framework MaCAPA mitigates high communication costs that are caused by low bandwidths in all cases. In particular, one mapping (for the benchmark *MPI2007/125.RAxML*, No. 33) has allocated all 8 processes to the dual-quad-core server (Node N_7), thus bypassing the slow Ethernet completely. For all other mappings, MaCAPA has excluded this server, which has extreme

low bandwidths to and from other servers. Thus, MaCAPA has automatically bypassed these low bandwidths.

In conclusion, our framework MaCAPA gains a large performance improvement of 35.9% on average for the MPI benchmarks if 8 processes execute each program. The run-time performance is degraded only in 10 cases by merely about 1 second on average. In contrast, the mean execution time reduction that MaCAPA achieves is 2182 seconds, i. e., about 36 minutes. In the following, we evaluate the experiments to assess the performance gain of MaCAPA if 16 processes execute a program.

Execution with 16 Processes

In Figure 7.16, we show the performance improvement that our framework MaCAPA achieves when 16 processes have to execute a program. As can be seen, the run-time performance is significantly improved for all but 2 of the benchmarks compared to the default MPI allocation strategy. For 19 out of 34 benchmarks, we have observed a huge performance gain by more than 60% and up to 99%. For 7 benchmarks, MaCAPA achieves a notable performance improvement by more than 20%. For only 6 benchmarks, we have observed smaller improvements between 4.6% and 20%, and a performance degradation solely for 2 benchmarks. On average, MaCAPA achieves a performance improvement of 59.4% across all benchmarks (see the rightmost bar).

Figure 7.17 shows the required execution times in seconds of each benchmark. Again, we compare the average execution time if the processes are allocated with the default MPI strategy against the time if MaCAPA determines

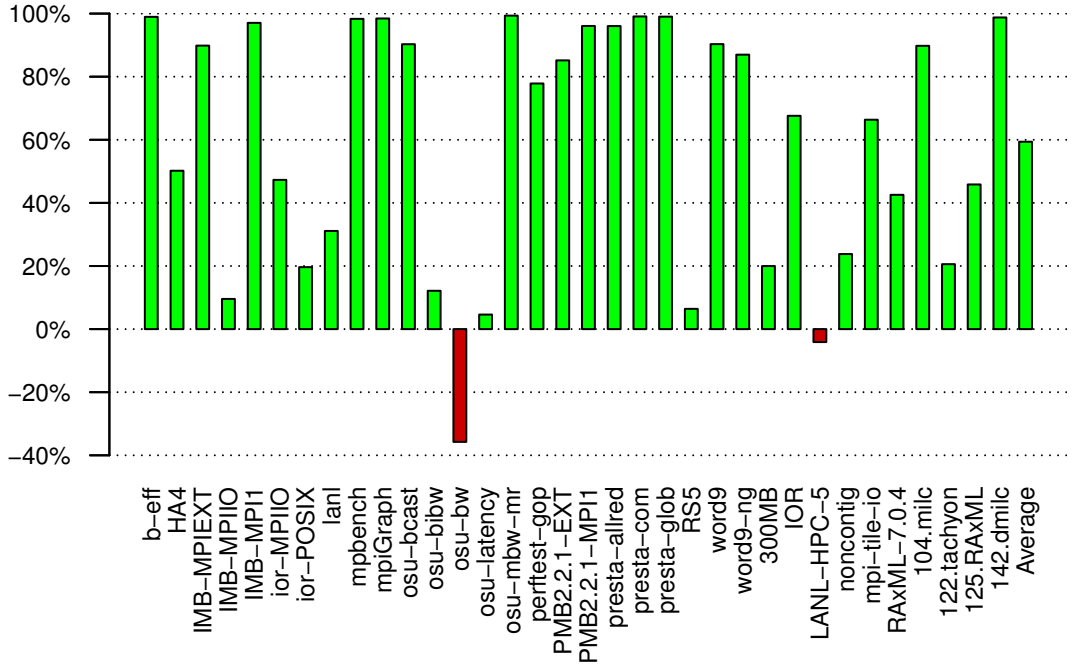


Figure 7.16: Performance – 16 processes: improvement gained with MaCAPA

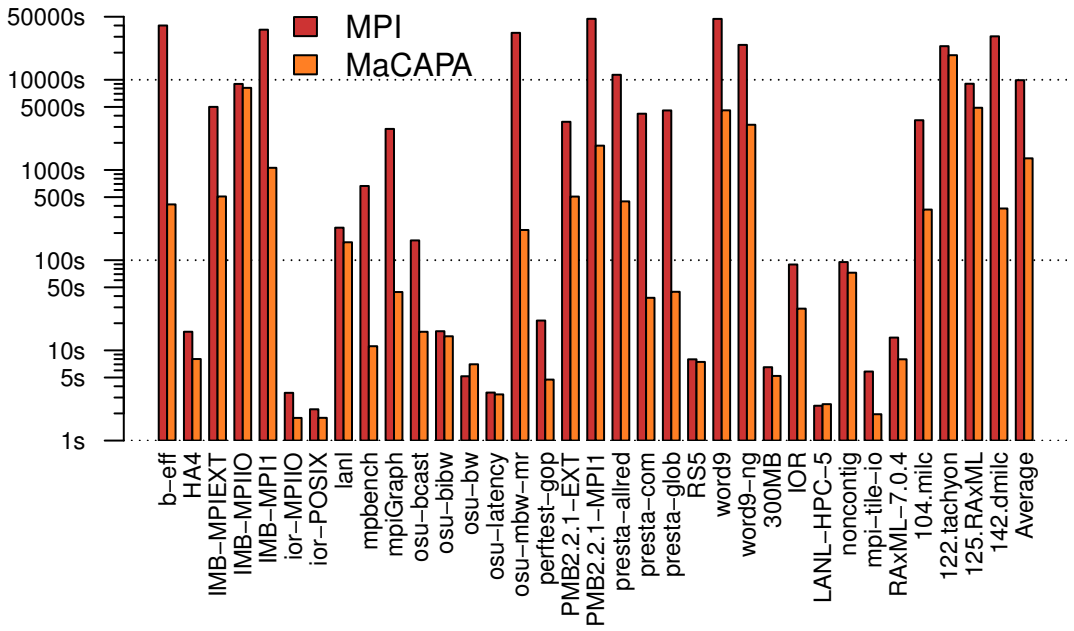


Figure 7.17: Performance – 16 processes: comparison between MPI and MaCAPA

the allocation. We see that the 2 benchmarks with a performance degradation again are those with one of the shortest execution times (note the logarithmic scale of the y-axis). For these 2 benchmarks, MaCAPA causes solely an increased execution time by 1 second on average. In contrast, the execution times are drastically reduced for the other benchmarks, with an average of 8552 seconds (about 142.5 minutes) across all benchmarks. MaCAPA achieves the greatest execution time reduction by 45572 seconds (about 12.7 hours) with the benchmark *PMB2.2.1-MPI1*.

In Table 7.10, we show the detailed execution times in seconds for the three runs of each benchmark. As can be seen, the execution times of the benchmarks across all three runs mostly are in orders of magnitudes smaller with the MaCAPA allocation compared to the default MPI allocation. We again see variations in execution times of the three runs with the MaCAPA allocation for the benchmarks 1–18, 25, 27, and 29, though they stem from identical invocations (since the benchmarks do not have configuration parameters). This must be caused by network traffic on the Ethernet. However, the variations in execution times with the default MPI strategy for these benchmarks do not stem from identical invocations. For each run, the processes are allocated to different PEs since no information about communication overheads is available. Thus, the default MPI strategy finds better or worse mappings merely by chance. A drastic example is the benchmark *Cbench/mpiGraph* (No. 10), where the *small data* run needs less than 1% of the execution time of the *medium data* run. In contrast, the mapping with MaCAPA considers communication costs and therefore has determined a good allocation. In particular, the Node N_7 is not considered as a candidate for the allocation because of its extreme low bandwidths to and from other nodes. For all but one bench-

Table 7.10: Performance – 16 processes: detailed execution times in seconds

No.	Benchmark	Small data		Medium data		Large data	
		MPI	MaCAPA	MPI	MaCAPA	MPI	MaCAPA
1	Cbench/b-eff	39169.58	415.57	42289.42	410.29	38513.00	420.06
2	Cbench/HA4	12.54	7.59	19.43	7.57	16.31	8.89
3	Cbench/IMB-MPIEXT	4282.56	485.72	3803.27	493.74	6934.71	545.91
4	Cbench/IMB-MPIIO	8923.60	7994.00	9655.00	8134.00	8418.80	8289.00
5	Cbench/IMB-MPI1	14095.42	1055.38	48376.56	1057.16	45308.36	1058.21
6	Cbench/ior-MPIIO	6.17	1.42	2.24	1.90	1.74	2.03
7	Cbench/ior-POSIX	2.29	1.41	2.50	1.91	1.88	2.04
8	Cbench/lanl	309.04	158.29	205.32	157.24	173.528	158.39
9	Cbench/mpbench	476.30	8.09	647.51	8.88	870.56	16.39
10	Cbench/mpiGraph	42.41	38.42	6246.33	38.68	2273.73	55.95
11	Cbench/osu-bcast	112.60	16.28	237.33	15.74	146.95	16.21
12	Cbench/osu-bibw	10.19	15.42	25.24	13.81	13.50	13.76
13	Cbench/osu-bw	4.78	7.53	5.34	6.85	5.38	6.66
14	Cbench/osu-latency	3.13	2.45	3.85	3.22	3.24	4.08
15	Cbench/osu-mbw-mr	32494.41	210.02	11393.29	210.58	55539.07	227.78
16	Cbench/perftest-gop	41.67	3.30	15.08	2.76	7.47	8.17
17	Cbench/PMB2.2.1-EXT	2207.91	486.75	2879.49	497.35	5176.33	538.10
18	Cbench/PMB2.2.1-MPI1	23688.66	1876.37	76280.60	1843.60	42331.44	1865.33
19	Cbench/presta-allred	28882.81	434.05	3355.43	433.60	1837.61	478.15
20	Cbench/presta-com	2282.28	10.56	2038.92	10.82	8303.45	93.40
21	Cbench/presta-glob	2679.34	11.88	1787.37	11.89	9251.08	110.10
22	Cbench/RS5	4.74	4.33	7.02	5.87	12.11	12.14
23	Cbench/word9	47862.38	2944.60	55445.40	2935.83	38753.27	7879.00
24	Cbench/word9-ng	15322.44	2179.14	49831.16	2193.15	7940.55	5145.00
25	Cbench/300MB	5.47	5.37	8.83	4.81	5.22	5.44
26	IOR	10.35	3.42	168.45	12.71	89.84	70.92
27	LANL-HPC-5	2.45	1.78	3.00	2.34	1.85	3.48
28	PIO/noncontig	4.40	4.14	19.97	20.92	261.48	192.80
29	PIO/mpi-tile-io	1.68	3.49	1.61	1.70	14.19	0.69
30	RAxML-7.0.4	29.69	11.95	7.63	5.81	4.27	6.13
31	MPI2007/104.milc	67.15	20.58	318.59	20.39	10297.49	1050.72
32	MPI2007/122.tachyon	3.37	2.26	27553.60	20408.00	43211.80	35795.00
33	MPI2007/125.RAxML	10881.80	3272.64	11559.60	5691.00	4682.40	5725.00
34	MPI2007/142.dmilc	3612.73	60.42	829.718	23.92	86578.80	1038.62

mark (*MPI2007/125.RAxML*), MaCAPA automatically has excluded this node from the allocation.

In conclusion, the average performance improvement gained by MaCAPA with 16 executing processes is even higher than with 8 processes. We are able

to automatically improve the run-time performance by 59.4% on average with our instantiated framework MaCAPA. For all but 2 benchmarks, MaCAPA achieves a performance improvement by up to 99%. For 2 benchmarks, the performance is degraded by merely 1 second on average (respectively 20%). In contrast, the average execution time reduction gained with MaCAPA is 8552 seconds, i. e., about 142.5 minutes. In the following, we summarize our experimental results.

7.4 Summary

In this chapter, we have evaluated the performance gain if our instantiated framework MaCAPA is used to map MPI processes to a heterogeneous network of processors. First, we have performed experiments to investigate the accuracy of ML based predictors, which our framework employs to establish precise cost models for the mappings. The predictors for unknown loop iteration counts and recursion frequencies of functions classify the regarded behavior. The experimental results demonstrate that the relationship between static code features and classes can successfully be learned with a mean absolute error of 0.72 in both cases and a correlation of 0.47 and 0.45, respectively. For learning execution times, we have shown that our adaption of the PQR2 technique outperforms other regression algorithms. The median deviation of our predictions from the actual times is 1.8 microseconds (the skewed mean is 16.8 seconds). Furthermore, above 85% of the predictions deviate by less than 250 milliseconds from the actual value. The correlation (0.12) is statistically significant such that we have 99.9% evidence that a relationship is learned. Based on execution time predictors, we have successfully learned computational performance differences between different CPU models. With the resulting predictor, we are able to exactly predict the best performing PE for 53% of the functions. This yields a mean performance improvement of 39.5% for sequential benchmarks.

In our main experiments to investigate the actual performance improvement caused by MaCAPA, we have shown that MaCAPA precisely identifies communication overheads. The experimental results if 8 MPI processes execute each benchmark demonstrate that the required execution time is reduced by about 36 minutes on average. This mean performance improvement by 35.9% with MPI benchmarks is similar to the mean performance improvement of 39.5% for learning the best performing PE with sequential benchmark suites. Hence, we have evidence that MaCAPA mitigates both the computational costs and the communication costs. The mean performance improvement that MaCAPA achieves if 16 processes execute each MPI benchmark is 59.4%, which is an execution time reduction by about 142.5 minutes on average. Overall, our experiments demonstrate the applicability of MaCAPA to huge software in the area of HPC, the accuracy of employed ML predictors, and the significant performance improvement gained with our MaCAPA framework.

8 Conclusion and Future Work

In this chapter, we summarize and discuss the central results of this thesis. We also review the criteria we have defined in the introduction and discuss whether they have been fulfilled. Then, we outline and discuss directions for potential future work.

8.1 Results

In this thesis, we have presented our general framework MaCAPA for *Machine Learning based **Ma**pping of **C**oncurrent **A**pplications to **P**arallel **A**rchitectures*. The framework aims at improving this mapping by providing a power-efficient and communication-aware allocation of the parallelly executable tasks of applications to the *Processing Elements* (*PEs*) of a heterogeneous target architecture. Thereby, our framework MaCAPA automatically optimizes the run-time performance of applications.

The basic idea of our framework is to statically determine computational costs and communication overheads for the parallelly executable tasks of a given program. To that end, our framework employs *Machine Learning* (*ML*) techniques to predict needed information about unknown run-time behavior. This knowledge is then used to derive a precise cost model w. r. t. the program in a given architectural setting, which makes it possible to automatically rate the gain of alternative mappings. We have presented a novel allocation algorithm that employs our ML based cost model and selects the mapping with the highest gain. Our static allocation heuristics predicts the best PE that is available, thereby determining whether it is possible to map more than one task to the same PE without performance degradation. By doing this, the power consumption of the target architecture can be reduced because other PEs can potentially be put in a low-power state. Our dynamic allocation heuristics considers the reallocation of tasks if it is beneficial and allocates conditionally created tasks at run-time on the PE with the minimal run-time cost under the current workload of the PEs. Hence, our mapping algorithm not only improves an initial static mapping, but also dynamically reduces run-time overheads, which is crucial for the resulting run-time performance of programs.

We have presented our approaches for learning iteration counts of loops and recursion frequencies of functions. This allows us to precisely predict communication costs between tasks (if communication arises within loops or recursive functions, respectively). We also have presented our approaches for learning execution times of tasks and, based on this, for learning the PE that executes a task most efficiently. This allows us to precisely predict computational costs of tasks on different PEs of the target architecture and, based on this, to rate the PEs according to their computational performance. The machine learned models are automatically transformed to heuristics that predict the regarded run-time behavior solely based on static code features of applications. Hence, it is only necessary to statically analyze a number of code features to obtain a prediction. Since the generated heuristics are automatically incorporated into the MaCAPA framework, there is no need for user intervention. Furthermore, no user annotations are required for needed information about unknown run-time behavior. Additionally, no profiling at compile time is necessary to derive information about expectable run-time behavior because learning is done in a one-off training phase per architecture decoupled from compilation. In conclusion, our ML knowledge establishes efficient and highly scalable predictors for regarded run-time behavior, which can be used for a power-efficient and communication-aware allocation of tasks to PEs of the target architecture. That is, our approach provides the fully automatic improvement of mappings of arbitrary concurrent applications to any parallel architecture.

The framework is applicable to various parallel programming models in combination with different target architectures. We have presented an instantiation of our general framework, which aims at improving the mapping of MPI programs to heterogeneous networks of processors. Reducing the communication overhead is highly important for processor networks since the communication medium has a much lower bandwidth than between a PE and the main memory (which can be used in a shared memory system). Hence, our instantiated framework mitigates the major performance limitation for applications executing on distributed memory architectures.

We have implemented the instantiated framework, whereby we have obtained a complete experimental platform with which we have assessed the actual performance improvement that can be achieved with our approach. In our experiments, we first have shown that the ML heuristics can precisely predict the regarded behavior and outperform other static approaches to predict the behavior of applications. Both the predictions for loop iteration counts and recursion frequencies of functions deviate from the actual (classified) behavior by 0.72 classes on average. The high correlation in both cases (0.47 and 0.45, respectively) demonstrates that the ML algorithms successfully have learned the relationship between static code features and dynamic run-time behavior. For learning execution times, we have shown that our approach based on a *Predicting Query Run-time 2* (*PQR2*) tree is best suited to statically predict execution times of tasks. Our approach yields a median absolute error of 1.8 microseconds and the predictions show a statistically significant correlation to the actual execution time (0.12). For learning the best performing PE, we have demonstrated that we correctly have predicted the best PE in 53% of the

cases and in 17% of the cases the second best PE to be the best, which gains a mean execution time reduction of 39.5% (for serial benchmarks when they are allocated to our predicted PEs).

In our main experiments with 34 MPI benchmarks, we have shown that our instantiated MaCAPA framework actually yields a significant performance gain. Compared to the default MPI allocation, MaCAPA reduces the average execution time by 35.9% on average when 8 processes execute each benchmark and by 59.4% when 16 processes execute each benchmark. The experiments with our large benchmark collection composed of real-world programs from various application domains furthermore indicate the general applicability of our MaCAPA framework to any application. In conclusion, we have made a contribution to the important parallel performance optimization problem, which rules many application domains.

8.2 Discussion

In the introducing chapter, we have defined a set of criteria that a framework for the mapping of concurrent applications to parallel architectures should meet. In the following, we review the objectives and discuss whether our framework MaCAPA fulfills the corresponding criteria.

The complete framework does not require user intervention and our analyses eliminate the requirement for manual annotations concerning needed run-time information. Hence, the *automatic applicability* is given. Our static analyses of concurrent tasks yield close approximations for the expectable run-time behavior. Hence, their *precision* is given (as far as statically possible). For our analyses of the run-time overhead, which employ ML techniques, we have experimentally demonstrated the precision of the learned predictors. Our framework ensures an *efficient compilation* since, on the one hand, the predictors are learned in a one-time-only training phase decoupled from compilation and, on the other hand, our analyses of the expectable behavior do not impose a larger compile time overhead than usual program analyses as used in compilers. The *scalability* of our ML techniques has been shown in general by their application to large databases, called data mining, and in particular by our experiments with more than 20,000 observations. Our mapping heuristics also scales for large task graphs since our greedy algorithm does not exhaustively explore the search space of possible allocations. Our MaCAPA framework extends a conventional compiler whose front end first transforms the source code to an IR from which finally machine code is generated in the back end of the compiler. The only prerequisite for the applicability of our approach is the existence of compilers for parallel programming languages (realizing different programming models) that can generate code for existing architectures. Since our framework does not impose other restrictions, it is *general* enough to cope with various *parallel programming models* and *target architectures*. The *generality of the cost model* is also given because our cost model employs hardware-dependent information about computational performance of PEs and

communication overheads caused from allocations on PEs such that it is able to rate the gain of mappings to different architectures. With our experiments to evaluate the performance gain that our instantiated framework achieves, we have shown that MaCAPA can significantly reduce the required execution time of MPI programs. We also have shown the *generality of application domains* since we have used a considerable number of real-world programs from various application domains for our experiments.

In conclusion, our MaCAPA framework provides automatic, efficient, and highly scalable methods for optimizing the run-time performance of arbitrary concurrent applications. Hence, MaCAPA fulfills all criteria that have to be met by a framework for the mapping of concurrent applications to parallel architectures.

8.3 Future Work

In this thesis, we have presented a general framework to improve the mapping of concurrent applications to parallel architectures. Furthermore, we have instantiated our framework to the MPI mapping problem targeting heterogeneous networks of processors. With our implemented experimental platform of the instantiated framework, we have demonstrated that we are able to achieve a significant performance gain with our approach. During our intensive work in the field of parallel systems, several interesting open questions have arisen that are worth to be investigated in further research. In the following, we first outline future work that extends our instantiated framework. Then, we discuss future research on our general framework. Finally, we describe research directions based upon our approach presented in this thesis.

Our instantiated framework automatically allocates MPI processes to available PEs in the processor network, i. e., to cores of the CPUs in the network. Our cost model considers, among others, the computational costs of processes. However, the cost model does not address the fact that a process may execute its own code concurrently (e. g., by spawning threads). A sensible extension of our instantiated framework is to adapt our cost model to this aspect. On the one hand, the allocation for such a process should target a CPU having multiple cores, and, on the other hand, more than one core of a CPU should be reserved for a concurrent process to enable a performant execution in parallel. Our implementation of the instantiated framework does not consider the reallocation of processes between different CPUs, as it is the case for the conceptual algorithm (it supports only the reallocation between cores of a CPU). It would be interesting whether the resulting performance gain is worth the effort to maintain execution states of tasks and to transfer necessary data for the reallocation.

Our general framework provides the mapping for applications realizing various parallel programming models. It would be worthwhile to apply our general framework to other models than the message passing model and to evaluate the performance gain this instantiated framework would cause. In particular,

a *hybrid* parallel programming model, e. g., a combination of OpenMP (which realizes the shared memory model) and MPI, is a promising candidate. Depending on the given architecture (e. g., a slow interconnection network), a hybrid programming model can be superior to the other models [JJaMH03, RHJ09]. Furthermore, it would also be interesting to target other architectures with our framework, for example heterogeneous *Multi-Processor Systems-on-Chip* (*MPSoCs*). By that, we could investigate whether our framework also achieves a significant performance gain for applications if the PEs of the architecture are connected with a high-speed network. Another attractive target for the instantiation would be an architecture that actually provides energy saving by putting a PE in a low power state or by *Dynamic Voltage and Frequency Scaling*. Then, we could measure the actual gain of our power-aware allocation concerning the energy consumption of applications. Our mapping algorithm exclusively considers a power-aware allocation if it does not degrade the performance. However, in some application domains, such as the embedded systems domain, the energy consumption may be more constrained than the computational performance. Hence, it can be worthwhile to develop a cost model that explicitly allows a performance degradation to reduce the energy consumption, preferably parameterized to weight both goals against each other.

The principal idea of our general framework is to employ ML based heuristics for establishing a precise cost model that rates the gain of alternative mappings. Considering our promising experimental results, we are convinced that our general idea – ML predictors for unknown information, a cost model for computational and communication costs to assess the quality of choices, and minimizing the value of an objective function – can be successfully transferred to other research domains. For example, modern embedded systems often require to execute multiple applications on the same resource. In this case, the automatic and efficient prediction of run-time overheads for *simultaneously* mapping *multiple concurrent applications* to a shared resource can be addressed with our approach. Likewise, automatically minimizing run-time overheads of applications for *dynamic load balancing* on a distributed system (i. e., if multiple users start applications not simultaneously) is still an open research question. A research domain of growing importance are *cyber-physical systems*, where a number of distributed embedded systems that monitor and control physical processes are interconnected via a network, for example the Internet. In this setting, a cyber-physical system integrates the physical processes such that it forms a closed system. At the same time, a variety of communication media is often used. Hence, cyber-physical systems are inherently concurrent, with restricted computation and communication resources. In this area, fundamental challenges remain to be solved, such as energy control, system resource allocation, reliability, safety, and security. We think that our approach can be used as starting point to tackle these problems. On the one hand, our general idea of generalizing the behavior from previous observations via ML is also applicable to predict the effect of physical processes. On the other hand, our cost model, which includes hardware-dependent information about computation and communication overheads, is well-suited for a network of computational elements. For the same reason, we

are convinced that our approach is also applicable to resource constrained *networked embedded systems* in general. An interesting research question arises from the dynamics of networks (i. e., where the number of nodes or communication channels may change). This is not perfectly handled with our approach since we propose a one-off ML phase per architecture and we then assume the architecture to be fixed. However, the phase may also be integrated by going from offline learning to *online learning*.

As short-term goal, we aim at the instantiation of the general framework for other programming models and target architectures, for example a hybrid programming model and heterogeneous MPSoCs. For the long-term, we are convinced that our approach can contribute to solving new research challenges concerning cyber-physical systems.

Bibliography

References

[AAJ+11] Eduardo Antunes, Alexandra Aguiar, F. Sergio Johann, Marcos Sartori, Fabiano Hessel, and César Marcon. Partitioning and mapping on NoC-based MPSoC: an energy consumption saving approach. In *Proceedings of the 4th International Workshop on Network on Chip Architectures*, NoCArc'11, pages 51–56, New York, NY, USA, 2011. ACM. ISBN: 978-1-4503-0947-9. http://dx.doi.org/10.1145/2076501.2076512.
(Referenced on pages 66 and 67.)

[AASA13] Mohammad Arjomand, S. Hamid Amiri, and Hamid Sarbazi-Azad. Efficient genetic based topological mapping using analytical models for on-chip networks. *Journal of Computer and System Sciences*, 79(4):492–513, 2013. http://dx.doi.org/10.1016/j.jcss.2012.09.014.
(Referenced on pages 66 and 67.)

[ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI'97, pages 85–96, New York, NY, USA, 1997. ACM. ISBN: 0-89791-907-6. http://dx.doi.org/10.1145/258915.258924.
(Referenced on pages 84, 85, and 151.)

[ACE13] ACE. *Associated Compiler Experts bv., Amsterdam, The Netherlands*, 2013. http://www.ace.nl.
(Referenced on pages 23 and 149.)

[ACP+95] Todd Austin, Raymond Chen, Dean Pomerleau, Alain Kägi, and Scott Breach. *The Pointer-intensive Benchmark Suite*, September 1995. http://pages.cs.wisc.edu/~austin/ptr-dist.html.
(Referenced on page 166.)

[AI13] AbsInt GmbH. *aiSee - Graph Visualization*, 2013. http://www.absint.com/aisee/.
(Referenced on page 160.)

[AJW+05] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce L. Jacob, Chau-Wen Tseng, and Donald Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'05, pages 2–9, Austin, Texas, USA, March 2005. http://dx.doi.org/10.1109/ISPASS.2005.1430554.
(Referenced on page 166.)

[Alp10] Ethem Alpaydin. *Introduction to Machine Learning, Second Edition.* Adaptive Computation and Machine Learning. The MIT Press, 2nd edition, 2010. ISBN: 978-0-262-01243-0.
(Referenced on pages 27, 31, and 36.)

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Pearson/Addison-Wesley, 2 edition, 2007. ISBN: 0-321-48681-1.
(Referenced on page 23.)

[ARPS06] G. Agosta, S. Crespi Reghizzi, P. Palumbo, and M. Sykora. Selective compilation via fast code analysis and bytecode tracing. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 906–911, New York, NY, USA, 2006. ACM. ISBN: 1-59593-108-2. http://dx.doi.org/10.1145/1141277.1141488.
(Referenced on page 56.)

[AS04] Todd M. Austin and SimpleScalar LLC. X benchmarks, 2004. http://www.simplescalar.com/.
(Referenced on page 166.)

[BB03] Jinbo Bi and Kristin P. Bennett. Regression error characteristic curves. In Tom Fawcett and Nina Mishra, editors, *Proceedings of the 20th International Conference on Machine Learning*, ICML, pages 43–50. AAAI Press, 2003. ISBN: 1-57735-189-4. http://www.aaai.org/Library/ICML/2003/icml03-009.php.
(Referenced on page 185.)

[BC04] Árpád Beszédes and Paolo Carlini. CSiBE benchmark: One year perspective and plans. In *Proceedings of GCC Developers' Summit*, pages 7–15, Ottawa, Canada, June 2004. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.1825.
(Referenced on page 166.)

[BCC+10] Moshe (Maury) Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-

Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. Analyzing parallel programs with Pin. *Computer*, 43(3):34–41, March 2010. http://dx.doi.org/10.1109/MC.2010.60.
(Referenced on page 151.)

[BDH+10] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, January 2010. http://dx.doi.org/10.3233/SPR-2009-0296.
(Referenced on page 44.)

[BFOS84] Leo Breiman, Jerome Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. The Wadsworth and Brooks-Cole statistics-probability series. Chapman & Hall, Monterey, CA, 1984. ISBN: 9780412048418.
(Referenced on pages 28, 36, and 92.)

[BK02] Gianluca Bontempi and Wido Kruijtzer. A data analysis method for software performance prediction. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE'02, pages 971–976, Washington, DC, USA, 2002. IEEE Computer Society. ISBN: 0-7695-1471-5. http://dx.doi.org/10.1109/DATE.2002.998417.
(Referenced on page 60.)

[BMPW12] Jennifer Brouner, Morgan McCorkle, Jim Pearce, and Leo Williams. Titan rising: the world's fastest computer for science. *Oak Ridge National Laboratory Review*, 45(3):1–28, 2012.
(Referenced on page 46.)

[BMYO07] Takanobu Baba, Tomohisa Masuho, Takashi Yokota, and Kanemitsu Ootsu. Design of a two-level hot path detector for path-based loop optimizations. In *Proceedings of the third conference on IASTED International Conference: Advances in Computer Science and Technology*, ACST'07, pages 23–28, Anaheim, CA, USA, 2007. ACTA Press. http://portal.acm.org/citation.cfm?id=1322468.1322472.
(Referenced on page 18.)

[Boe10] Hans Boehm. An artificial garbage collection benchmark, 2010. http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench.html.
(Referenced on page 166.)

[BPP07] Debasish Basak, Srimanta Pal, and Dipak Chandra Patranabis. Support Vector Regression. *Neural Information Processing – Letters and Reviews*, 11(10):203–224, October 2007.
(Referenced on page 36.)

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. http://dx.doi.org/10.1023/A:1010933404324.

(Referenced on pages 30, 91, 93, 155, and 169.)

[BW09] Raymond P. L. Buse and Westley Weimer. The road not taken: Estimating path execution frequency statically. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE'09, pages 144–154, Washington, DC, USA, 2009. IEEE Computer Society. ISBN: 978-1-4244-3453-4. http://dx.doi.org/10.1109/ICSE.2009.5070516.
(Referenced on pages 52 and 57.)

[BZM01] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'01, pages 114–124, New York, NY, USA, 2001. ACM. ISBN: 1-58113-414-2. http://dx.doi.org/10.1145/378795.378821.
(Referenced on page 166.)

[CBG01] Matteo Corti, Roberto Brega, and Thomas Gross. Approximation of worst-case execution time for preemptive multitasking systems. In Jack Davidson and Sang Min, editors, *Languages, Compilers, and Tools for Embedded Systems*, volume 1985 of *Lecture Notes in Computer Science*, pages 178–198. Springer Berlin / Heidelberg, 2001.
(Referenced on pages 52, 53, 57, and 59.)

[CDSL11] John T. Criswell, Daniel Dunbar, Reid Spencer, and Tanya Lattner. LLVM Testing Infrastructure, 2011. http://llvm.org/docs/TestingGuide.html.
(Referenced on page 166.)

[CH67] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, January 1967. http://dx.doi.org/10.1109/TIT.1967.1053964.
(Referenced on page 34.)

[CHD11] Thidapat Chantem, Xiaobo Sharon Hu, and Robert P. Dick. Temperature-aware scheduling and assignment for hard real-time applications on MPSoCs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(10):1884–1897, October 2011. http://dx.doi.org/10.1109/TVLSI.2010.2058873.
(Referenced on pages 66 and 67.)

[Che05] R.J. Chevance. *Server Architecture: Multiprocessors, Clusters, Parallel Systems, Web Servers, Storage Units*. ITPro collection. Digital Press [Imprint], 2005. ISBN: 9781555583330.
(Referenced on page 43.)

[Cil01] Supercomputing Technologies Group and Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, November 2001. http://supertech.lcs.

mit.edu/cilk/manual-5.4.6.pdf.
(Referenced on page 41.)

[CKR+12] Sudipta Chattopadhyay, Chong Lee Kee, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified WCET analysis framework for multi-core platforms. In *Proceedings of the IEEE 18th Real-Time and Embedded Technology and Applications Symposium*, RTAS'12, pages 99–108. IEEE, April 2012. ISBN: 978-1-4673-0883-0. http://dx.doi.org/10.1109/RTAS.2012.26.
(Referenced on page 66.)

[CLA13] Jerónimo Castrillón, Rainer Leupers, and Gerd Ascheid. MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs. *IEEE Transactions on Industrial Informatics*, 9(1):527–545, 2013. http://dx.doi.org/10.1109/TII.2011.2173941.
(Referenced on pages 66 and 67.)

[CM07] Christoph Cullmann and Florian Martin. Data-flow based detection of loop bounds. In Christine Rochange, editor, *Proceedings of the 7th International Workshop on Execution-Time Analysis*, WCET'07, Dagstuhl, Germany, July 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. http://dx.doi.org/10.4230/OASIcs.WCET.2007.1193.
(Referenced on page 53.)

[CM11] Chen-Ling Chou and Radu Marculescu. FARM: Fault-aware resource management in NoC-based multiprocessor platforms. In *Proceedings of the 14th IEEE Conference on Design, Automation and Test in Europe*, DATE'11, pages 673–678, 2011. ISBN: 978-1-61284-208-0. http://dx.doi.org/10.1109/DATE.2011.5763113.
(Referenced on page 67.)

[Cor13] Paulo Cortez. *rminer: Simpler use of data mining methods (e. g. NN and SVM) in classification and regression*, 2013. http://CRAN.R-project.org/package=rminer. R package version 1.3.
(Referenced on page 150.)

[CR95] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'95, pages 29–38, New York, NY, USA, 1995. ACM. ISBN: 0-89791-700-6. http://dx.doi.org/10.1145/209936.209941.
(Referenced on page 166.)

[CR13] Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via path-sensitive ver-

ification. *Real-Time Systems*, pages 1–46, 2013. http://dx.doi.org/10.1007/s11241-013-9178-0.
(Referenced on page 66.)

[CRW07] Ayse Kivilcim Coskun, Tajana Simunic Rosing, and Keith Whisnant. Temperature aware task scheduling in MPSoCs. In Rudy Lauwereins and Jan Madsen, editors, *Proceedings of the 10th IEEE Conference on Design, Automation and Test in Europe*, DATE'07, pages 1659–1664. ACM, April 2007. ISBN: 978-3-9810801-2-4. http://dx.doi.org/10.1145/1266366.1266730.
(Referenced on pages 66 and 67.)

[CSACR09] Simone Campanoni, Martino Sykora, Giovanni Agosta, and Stefano Crespi Reghizzi. Dynamic look ahead compilation: A technique to hide JIT compilation latencies in multicore environment. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 220–235, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN: 978-3-642-00721-7. http://dx.doi.org/10.1007/978-3-642-00722-4_16.
(Referenced on page 56.)

[CVH+13] Dehao Chen, Neil Vachharajani, Robert Hundt, Xinliang D. Li, Stéphane Eranian, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for precise and versatile feedback directed optimizations. *IEEE Transactions on Computers*, 62(2):376–389, 2013. http://dx.doi.org/10.1109/TC.2011.233.
(Referenced on pages 14 and 58.)

[Dar01] Frederica Darema. The spmd model: Past, present and future. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131, pages 1–1. Springer Berlin / Heidelberg, 2001. ISBN: 3-540-42609-4. http://dx.doi.org/10.1007/3-540-45417-9_1.
(Referenced on page 40.)

[DB00] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-IX, pages 202–211, New York, NY, USA, 2000. ACM. ISBN: 1-58113-317-0. http://dx.doi.org/10.1145/378993.379241.
(Referenced on page 18.)

[DCF+07] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O'Boyle, and Olivier Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th International Conference on Com-*

puting Frontiers, CF'07, pages 131–142, New York, NY, USA, 2007. ACM. ISBN: 978-1-59593-683-7. http://dx.doi.org/10.1145/1242531.1242553.
(Referenced on page 49.)

[DJB+09] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, Grigori Fursin, and Michael F. P. O'Boyle. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 78–88, New York, NY, USA, 2009. ACM. ISBN: 978-1-60558-798-1. http://dx.doi.org/10.1145/1669112.1669124.
(Referenced on page 50.)

[DKV13a] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. Communication and migration energy aware design space exploration for multicore systems with intermittent faults. In *Proceedings of the 16th IEEE Conference on Design, Automation and Test in Europe*, DATE'13, pages 1631–1636, March 2013.
(Referenced on pages 66 and 67.)

[DKV13b] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems. In *Proceedings of the 16th IEEE Conference on Design, Automation and Test in Europe*, DATE'13, pages 689–694, March 2013.
(Referenced on page 67.)

[DL03] Yonghua Ding and Zhiyuan Li. A compiler analysis of interprocedural data communication. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 1–11, Washington, DC, USA, 2003. IEEE Computer Society. ISBN: 1-58113-695-1. http://dx.doi.org/10.1109/SC.2003.10009.
(Referenced on page 57.)

[DLR+07] Emiliano Dolif, Michele Lombardi, Martino Ruggiero, Michela Milano, and Luca Benini. Communication-aware stochastic allocation and scheduling framework for conditional task graphs in multi-processor systems-on-chip. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, EMSOFT'07, pages 47–56, New York, NY, USA, 2007. ACM. ISBN: 978-1-59593-825-1. http://dx.doi.org/10.1145/1289927.1289940.
(Referenced on pages 66 and 67.)

[Dob02] A.J. Dobson. *An Introduction to Generalized Linear Models, Second Edition*. Texts in Statistical Science Series. Taylor & Francis Group, 2002. ISBN: 9781584881650.
(Referenced on page 33.)

[dW02] Rob F. Van der Wijngaard. NAS parallel benchmarks version 2.4. NAS Technical Report NAS-02-007, NASA Ames Research Center, Moffett Field, CA, USA, October 2002. http://www.nas.nasa.gov/resources/software/npb.html.
(Referenced on page 166.)

[EG97] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of the Third International European Conference on Parallel Processing*, Euro-Par'97, pages 1298–1307, Passau, Germany, August 1997. Springer Berlin Heidelberg. ISBN: 3-540-63440-1. http://dx.doi.org/10.1007/BFb0002886.
(Referenced on pages 53 and 54.)

[ESG+07] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Christine Rochange, editor, *7th International Workshop on Worst-Case Execution Time Analysis*, WCET'07, Dagstuhl, Germany, July 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. http://dx.doi.org/10.4230/OASIcs.WCET.2007.1194.
(Referenced on page 54.)

[FDM+08] Guofu Feng, Xiaoshe Dong, Siyuan Ma, Jinghua Feng, and Xuhao Wang. A profile-based memory access optimizing technology on CBE architecture. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications*, HPCC'08, pages 382–388, Los Alamitos, CA, USA, 2008. IEEE Computer Society. ISBN: 978-0-7695-3352-0. http://dx.doi.org/10.1109/HPCC.2008.134.
(Referenced on pages 18, 56, and 67.)

[Fer11] Carlo Alberto Ferraris. *Compiler optimizations based on call-graph flattening*. PhD thesis, Third School of Engineering: Information Technology, Politecnico di Torino, Italy, 2011.
(Referenced on page 24.)

[FGMS91] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer. Availability of f2c – a Fortran to C converter. *SIGPLAN Fortran Forum*, 10(2):14–15, July 1991. http://dx.doi.org/10.1145/122006.122007.
(Referenced on page 168.)

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972. http://dx.doi.org/10.1109/TC.1972.5009071.
(Referenced on page 39.)

[FMT+08] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, François Bodin, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, and Michael O'Boyle. MILEPOST GCC: Machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, pages 7–19, June 2008.
(Referenced on pages 49 and 50.)

[Fos95] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. `http://portal.acm.org/citation.cfm?id=527029`. ISBN: 0201575949.
(Referenced on page 41.)

[FSTW09] Jason E. Fritts, Frederick W. Steiling, Joseph A. Tucek, and Wayne Wolf. Mediabench II video: Expediting the next generation of video systems research. *Microprocessors and Microsystems*, 33(4):301–318, 2009. `http://dx.doi.org/10.1016/j.micpro.2009.02.010`.
(Referenced on page 166.)

[Ful04] Brent Fulgham. Revived version of Bagley's great computer language shootout benchmarks, 2004. `http://shootout.alioth.debian.org/`.
(Referenced on page 166.)

[Fur09] Grigori Fursin. Collective tuning initiative: Automating and accelerating development and optimization of computing systems. In *Proceedings of the GCC Developers' Summit*, June 2009. `http://www.ctuning.org/cbench`.
(Referenced on page 166.)

[GDTF+12] Jorge González-Domínguez, Guillermo L. Taboada, Basilio B. Fraguela, María J. Martín, and Juan Touriño. Automatic mapping of parallel applications on multicore architectures using the servet benchmark suite. *Computers & Electrical Engineering*, 38(2):258–269, 2012. `http://dx.doi.org/10.1016/j.compeleceng.2011.12.007`.
(Referenced on pages 66 and 67.)

[GESL06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS'06, pages 57–66, Los Alamitos, CA, USA, December 2006. IEEE Computer Society. ISSN: 1052-8725. `http://dx.doi.org/10.1109/RTSS.2006.12`.
(Referenced on page 54.)

[GGL12] Andreas Gustavsson, Jan Gustafsson, and Björn Lisper. Toward static timing analysis of parallel software. In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis,* WCET'12, volume 23 of *OASIcs*, pages 38–47. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2012. ISBN: 978-3-939897-41-5. http://dx.doi.org/10.4230/OASIcs.WCET.2012.38.
(Referenced on page 66.)

[GKT+09] William Gropp, Ken Kennedy, Linda Torczon, Andy White, Jack Dongarra, Ian Foster, and Geoffrey C. Fox. *Sourcebook of parallel computing.* Morgan Kaufmann, 2009. ISBN: 1558608710.
(Referenced on page 41.)

[GLM04] Robert B. Gramacy, Herbert K. H. Lee, and William G. Macready. Parameter space exploration with Gaussian process trees. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, pages 45–53, New York, NY, USA, 2004. ACM. ISBN: 1-58113-838-5. http://dx.doi.org/10.1145/1015330.1015367.
(Referenced on page 28.)

[GMD08] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. PQR: Predicting query execution times for autonomous workload management. In *Proceedings of the 2008 International Conference on Autonomic Computing*, ICAC'08, pages 13–22, Washington, DC, USA, June 2008. IEEE Computer Society. ISBN: 978-0-7695-3175-5. http://dx.doi.org/10.1109/ICAC.2008.12.
(Referenced on pages 37 and 63.)

[GMH01] Paolo Giusto, Grant Martin, and Ed Harcourt. Reliable estimation of execution time of embedded software. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE'01, pages 580–589, Piscataway, NJ, USA, 2001. IEEE Press. ISBN: 0-7695-0993-2. http://dl.acm.org/citation.cfm?id=367072.367827.
(Referenced on page 59.)

[GRCG08] Francesc Guim, Ivan Rodero, Julita Corbalan, and Ariel Goyeneche. The grid backfilling: a multi-site scheduling architecture with data mining prediction techniques. In *Grid Middleware and Services*, pages 137–152. Springer US, 2008. http://dx.doi.org/10.1007/978-0-387-78446-5_10. ISBN: 978-0-387-78445-8.
(Referenced on page 62.)

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 2001 IEEE International Workshop on Workload Characterization*, WWC-4, pages 3–14, 2001.

(Referenced on pages 50 and 166.)

[GS85] Huisheng Gong and Monika Schmidt. A complexity measure based on selection and nesting. *SIGMETRICS Perform. Eval. Rev.*, 13:14–19, June 1985. http://dx.doi.org/10.1145/1041838.1041840.
(Referenced on pages 74 and 75.)

[GSH09] Pavel Ghosh, Arunabha Sen, and Alexander Hall. Energy efficient application mapping to NoC processing elements operating at multiple voltage levels. In *Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, NOCS'09, pages 80–85, Washington, DC, USA, 2009. IEEE Computer Society. ISBN: 978-1-4244-4142-6. http://dx.doi.org/10.1109/NOCS.2009.5071448.
(Referenced on pages 66 and 67.)

[GT13] Gary Grider and Alfred Torrez. *MPI-FTW benchmark*. Los Alamos National Laboratory, USA, 2013. http://institutes.lanl.gov/data/software/.
(Referenced on page 168.)

[GTT09] C. Goh, E. Teoh, and K. Tan. A hybrid evolutionary approach for heterogeneous multiprocessor scheduling. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 13:833–846, 2009. http://dx.doi.org/10.1007/s00500-008-0356-2.
(Referenced on page 67.)

[Gun98] Steve R. Gunn. Support vector machines for classification and regression. Technical report, University of Southampton, U.K., 1998.
(Referenced on page 35.)

[Gus07] Jan Gustafsson. The WCET tool challenge 2006. In Bernhard Steffen Tiziana Margaris, Anna Philippou, editor, *Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods*, ISOLA'06, pages 248–249, November 2007. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.
(Referenced on pages 54 and 166.)

[Gus10] Per Gustafsson. Bit stream benchmarks, 2010. http://user.it.uu.se/~pergu/bitbench/index.html.
(Referenced on page 166.)

[GZH93] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI'93, pages 177–186, New York, NY, USA, 1993. ACM. ISBN: 0-89791-598-4. http://dx.doi.org/10.1145/155090.155107.

(Referenced on page 166.)

[GZM+12] Yang Ge, Yukan Zhang, Parth Malani, Qing Wu, and Qinru Qiu. Low power task scheduling and mapping for applications with conditional branches on heterogeneous multi-processor system. *Journal of Low Power Electronics*, 8(5):535–551, 2012. http://dx.doi.org/10.1166/jolpe.2012.1214.
(Referenced on pages 66 and 67.)

[HB07] Simon Harding and Wolfgang Banzhaf. Fast genetic programming on GPUs. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 90–101, Valencia, Spain, April 11-13 2007. Springer. ISBN: 3-540-71602-5. http://dx.doi.org/doi:10.1007/978-3-540-71605-1_9.
(Referenced on page 44.)

[HBRK11] Jia Huang, Christian Buckl, Andreas Raabe, and Alois Knoll. Energy-aware task allocation for network-on-chip based heterogeneous multiprocessor systems. In *Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP'11, pages 447–454, Washington, DC, USA, 2011. IEEE Computer Society. ISBN: 978-0-7695-4328-4. http://dx.doi.org/10.1109/PDP.2011.10.
(Referenced on pages 66 and 67.)

[Hen00] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000. http://dx.doi.org/10.1109/2.869367.
(Referenced on page 166.)

[Hen06] John L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006. http://dx.doi.org/10.1145/1186736.1186737.
(Referenced on page 166.)

[HHS01] William W. Hargrove, Forrest M. Hoffman, and Thomas Sterling. The do-it-yourself supercomputer. *Scientific American*, 265(2):72–79, August 2001. http://www.sciam.com/article.cfm?articleID=000E238B-33EC-1C6F-84A9809EC588EF21.
(Referenced on page 46.)

[HJY+10] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In John D. Lafferty, Christopher K. I. Williams, John Shawe-Taylor, Richard S. Zemel, and Aron Culotta, editors, *Proceedings of the 24th Annual Conference on Neural Information Processing Systems*, NIPS'10, pages 883–891. Curran Associates, Inc., 2010. http://dblp.

uni-trier.de/db/conf/nips/nips2010.html#HuangJYCMN10.
(Referenced on page 64.)

[HNB+12] Miaoqing Huang, Vikram K. Narayana, Mohamed Bakhouya, Jaafar Gaber, and Tarek A. El-Ghazawi. Efficient mapping of task graphs onto reconfigurable hardware using architectural variants. *IEEE Transactions on Computers*, 61(9):1354–1360, September 2012. http://dx.doi.org/10.1109/TC.2011.153.
(Referenced on page 67.)

[HNY12] M. Hosseinabady and J.L. Nunez-Yanez. Run-time stochastic task mapping on a large scale network-on-chip with dynamically reconfigurable tiles. *IET Computers & Digital Techniques*, 6(1):1–11, 2012. http://dx.doi.org/10.1049/iet-cdt.2010.0097.
(Referenced on pages 66 and 67.)

[HP98] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Proceedings of the 5th International Symposium on Static Analysis*, SAS'98, pages 57–81, London, UK, 1998. Springer-Verlag. ISBN: 3-540-65014-8. http://portal.acm.org/citation.cfm?id=647167.717992.
(Referenced on page 166.)

[HR92] Todd Heywood and Sanjay Ranka. A practical hierarchical model of parallel computation. I. the model. *Journal of Parallel Distributed Computing*, 16(3):212–232, 1992. http://dx.doi.org/10.1016/0743-7315(92)90034-K.
(Referenced on page 39.)

[HSK+06] Y. Hotta, M. Sato, H. Kimura, S. Matsuoka, T. Boku, and D. Takahashi. Profile-based optimization of power performance by using dynamic voltage scaling on a PC cluster. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, volume 0 of *IPDPS'06*, pages 340–, Los Alamitos, CA, USA, 2006. IEEE Computer Society. ISBN: 1-4244-0054-6. http://dx.doi.org/10.1109/IPDPS.2006.1639597.
(Referenced on pages 18 and 56.)

[HSR+00] Christopher A. Healy, Mikael Sjödin, Viresh Rustagi, David B. Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems, The International Journal of Time-Critical Computing Systems*, 18(2-3):129–156, 2000. http://dx.doi.org/10.1023/A:1008189014032.
(Referenced on pages 52 and 53.)

[HTM10] Adam S. Hartman, Donald E. Thomas, and Brett H. Meyer. A case for lifetime-aware task mapping in embedded chip multiprocessors. In *Proceedings of the eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Sys-*

tem Synthesis, CODES/ISSS'10, pages 145–154, New York, NY, USA, 2010. ACM. ISBN: 978-1-60558-905-3. http://dx.doi.org/10.1145/1878961.1878987.
(Referenced on page 67.)

[HWC04] J. Henkel, W. Wolf, and S. Chakradhar. On-chip networks: a scalable, communication-centric embedded system design paradigm. In *17th International Conference on VLSI Design*, pages 845–851, 2004. http://dx.doi.org/10.1109/ICVD.2004.1261037.
(Referenced on page 45.)

[HYX11] Lin Huang, Feng Yuan, and Qiang Xu. On task allocation and scheduling for lifetime extension of platform-based MPSoC designs. *IEEE Transactions on Parallel and Distributed Systems*, 22(12):2088–2099, December 2011. http://dx.doi.org/10.1109/TPDS.2011.132.
(Referenced on page 67.)

[IEE04] Standard for information technology - portable operating system interface (POSIX). Shell and utilities. *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Shell and Utilities*, 2004. http://dx.doi.org/10.1109/IEEESTD.2004.94572.
(Referenced on page 41.)

[Ind04] Piotr Indyk. Nearest neighbors in high-dimensional spaces. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 39, pages 877–892. CRC Press LLC, Boca Raton, FL, 2nd edition, April 2004.
(Referenced on page 28.)

[IOF96] Michael A. Iverson, Füsun Özgüner, and Gregory J. Follen. Run-time statistical estimation of task execution times for heterogeneous distributed computing. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, HPDC '96, pages 263–270, Washington, DC, USA, August 1996. IEEE Computer Society. ISBN: 0-8186-7582-9. http://dx.doi.org/10.1109/HPDC.1996.546196.
(Referenced on page 58.)

[IOP99] Michael A. Iverson, Füsun Özgüner, and Lee C. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. *IEEE Transactions on Computers*, 48(12):1374–1379, December 1999. http://dx.doi.org/10.1109/12.817403.
(Referenced on page 58.)

[JJaMH03] G. Jost, H. Jin, D. an Mey, and F.F. Hatay. Comparing the OpenMP, MPI, and hybrid programming paradigms on an SMP cluster. In *Proceedings of the Fifth European Workshop on OpenMP*, EWOMP'03, 2003.
(Referenced on pages 42 and 205.)

[JKEM12] Olivera Jovanovic, Nils Kneuper, Michael Engel, and Peter Marwedel. ILP-based memory-aware mapping optimization for MPSoCs. In *Proceedings of the 15th IEEE International Conference on Computational Science and Engineering*, CSE'12, pages 413–420, Paphos, Cyprus, 2012. IEEE Computer Society. ISBN: 978-1-4673-5165-2. http://dx.doi.org/10.1109/ICCSE.2012.64.
(Referenced on page 67.)

[JMBT12] Olivera Jovanovic, Peter Marwedel, Iuliana Bacivarov, and Lothar Thiele. MAMOT: Memory-aware mapping optimization tool for MPSoC. In *Proceedings of the 15th Euromicro Conference on Digital System Design*, DSD'12, pages 743–750, September 2012. ISBN: 978-1-4673-2498-4. http://dx.doi.org/10.1109/DSD.2012.83.
(Referenced on page 67.)

[Kar72] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
(Referenced on pages 13 and 47.)

[KBDV08] Minyoung Kim, Sudarshan Banerjee, Nikil Dutt, and Nalini Venkatasubramanian. Energy-aware cosynthesis of real-time multimedia applications on MPSoCs using heterogeneous scheduling policies. *ACM Transactions on Embedded Computing Systems*, 7(2):9–:1–9:19, January 2008. http://dx.doi.org/10.1145/1331331.1331333.
(Referenced on page 45.)

[KGSC01] Chandra J. Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8):717–738, 2001. http://dx.doi.org/10.1002/spe.384.
(Referenced on page 56.)

[KKJ+08] Seongnam Kwon, Yongjoo Kim, Woo-Chul Jeun, Soonhoi Ha, and Yunheung Paek. A retargetable parallel-programming framework for MPSoC. *ACM Trans. Des. Autom. Electron. Syst.*, 13(3):1–18, 2008. http://dx.doi.org/10.1145/1367045.1367048.
(Referenced on page 42.)

[KKW+12] Joanna Kołodziej, Samee Ullah Khan, Lizhe Wang, Marek Kisiel-Dorohinicki, Sajjad A. Madani, Ewa Niewiadomska-Szynkiewicz,

Albert Y. Zomaya, and Cheng-Zhong Xu. Security, energy, and performance-aware resource allocation mechanisms for computational grids. *Future Generation Computer Systems*, 2012. http://dx.doi.org/10.1016/j.future.2012.09.009.
(Referenced on pages 66 and 67.)

[KKZ12] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for WCET analysis. In Edmund Clarke, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, volume 7162 of *Lecture Notes in Computer Science*, pages 227–242. Springer Berlin Heidelberg, 2012. http://dx.doi.org/10.1007/978-3-642-29709-0_20. ISBN: 978-3-642-29708-3.
(Referenced on page 55.)

[Kno08] Jens Knoop. Data-flow analysis for multi-core computing systems: A reminder to reverse data-flow analysis. In Florian Martin, Hanne Riis Nielson, Claudio Riva, and Markus Schordan, editors, *Scalable Program Analysis*, number 08161 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. ISSN: 1862-4405. http://drops.dagstuhl.de/opus/volltexte/2008/1575.
(Referenced on page 123.)

[KP84] Peter Kirschenhofer and Helmut Prodinger. Recursion depth analysis for special tree traversal algorithms. In Jan Paredaens, editor, *Automata, Languages and Programming*, volume 172 of *Lecture Notes in Computer Science*, pages 303–311. Springer Berlin / Heidelberg, 1984.
(Referenced on page 57.)

[KP01] Subhash Khot and Ashok Kumar Ponnuswami. Improved inapproximability results for maxclique, chromatic number and approximate graph coloring. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 600–609, 2001.
(Referenced on page 47.)

[Kra12] William T.C. Kramer. Top500 versus sustained performance: the top problems with the top500 list – and what to do about them. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 223–230, New York, NY, USA, 2012. ACM. ISBN: 978-1-4503-1182-3. http://dx.doi.org/10.1145/2370816.2370850.
(Referenced on page 46.)

[LCFM09] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'09, pages 136–146, Seattle, Washington, USA, March 2009. IEEE Computer Soci-

ety. ISBN: 978-0-7695-3576-0. http://dx.doi.org/10.1109/CGO.2009.17.
(Referenced on pages 53, 54, and 66.)

[LCLL09] Yuet Ming Lam, José Gabriel de Figueiredo Coutinho, Wayne Luk, and Philip Heng Wai Leong. Optimising multi-loop programs for heterogeneous computing systems. In *5th Southern Conference on Programmable Logic*, SPL'09, pages 129–134. IEEE, April 2009. http://dx.doi.org/10.1109/SPL.2009.4914904.
(Referenced on page 55.)

[LES+13] Björn Lisper, Andreas Ermedahl, Dietmar Schreiner, Jens Knoop, and Peter Gliwa. Practical experiences of applying source-level WCET flow analysis to industrial code. *International Journal on Software Tools for Technology Transfer*, 15:53–63, 2013. http://dx.doi.org/10.1007/s10009-012-0255-9.
(Referenced on page 66.)

[LFF07] Shun Long, Grigori Fursin, and Björn Franke. A cost-aware parallel workload allocation approach based on machine learning techniques. In *Proceedings of the IFIP International Conference on Network and Parallel Computing,* NPC'07, number 4672 in LNCS, pages 506–515. Springer Verlag, September 2007.
(Referenced on page 51.)

[LGMM09] Paul Lokuciejewski, Fatih Gedikli, Peter Marwedel, and Katharina Morik. Automatic WCET reduction by machine learning based heuristics for function inlining. In *Proceedings of the 3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation*, SMART'09, pages 1–15, Paphos, Cyprus, January 2009. http://unidapt.org/dissemination/workshops/smart09/papers/paper1.pdf.
(Referenced on page 50.)

[Liu07] ShinMing Liu. Open64 release 4.0: High performance compiler for Itanium and x86 Linux. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'07, 2007. http://www.open64.net/. © 2006 – 2012 Computer Architecture and Parallel Systems Laboratory.
(Referenced on page 23.)

[LMK08] Seong-Won Lee, Soo-Mook Moon, and Seong-Moo Kim. Enhanced hot spot detection heuristics for embedded java just-in-time compilers. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '08, pages 13–22, New York, NY, USA, 2008. ACM. ISBN: 978-1-60558-104-0. http://dx.doi.org/10.1145/1375657.1375660.
(Referenced on page 56.)

[LMM13] William Loewe, Tyce McLarty, and Christopher Morrone. *IOR HPC Benchmark*. Lawrence Livermore National Laboratory, University of California, USA, 2013. `http://sourceforge.net/projects/ior-sio/`.
(Referenced on page 168.)

[LOW09] Hugh Leather, Michael O'Boyle, and Bruce Worton. Raced profiles: efficient selection of competing compiler optimizations. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES'09, pages 50–59, New York, NY, USA, 2009. ACM. ISBN: 978-1-60558-356-3. `http://dx.doi.org/10.1145/1542452.1542460`.
(Referenced on page 50.)

[LPF+11] Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. Approximating Pareto optimal compiler optimization sequences – a trade-off between WCET, ACET, and code size. *Software: Practice and Experience*, 41(12):1437–1458, 2011. `http://dx.doi.org/10.1002/spe.1079`.
(Referenced on page 50.)

[LPMS97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO'30, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society. ISBN: 0-8186-7977-8. `http://portal.acm.org/citation.cfm?id=266800.266832`.
(Referenced on page 166.)

[LR97] William Landi and Barbara G. Ryder. Programming Languages Research Group, The State University of New Jersey, 1997. `http://prolangs.cs.vt.edu/rutgers/software.php`.
(Referenced on page 166.)

[LRN88] Joseph Lee Rodgers and W. Alan Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66, 1988. `http://dx.doi.org/10.1080/00031305.1988.10475524`.
(Referenced on pages 169, 182, and 187.)

[LSMM10] Paul Lokuciejewski, Marco Stolpe, Katharina Morik, and Peter Marwedel. Automatic selection of machine learning models for WCET-aware compiler heuristic generation. In *Proceedings of the 4th Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation*, SMART'10, pages 3–17, Pisa, Italy, January 2010. `http://ctuning.org/dissemination/smart10-01.pdf`.
(Referenced on page 50.)

[LW02] Andy Liaw and Matthew Wiener. Classification and regression by randomForest. *R News*, 2(3):18–22, 2002. `http://CRAN.R-project.org/doc/Rnews/`.
(Referenced on page 150.)

[LYGY10] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, RTSS'10, pages 339–349. IEEE, 2010. ISSN: 1052-8725. `http://dx.doi.org/10.1109/RTSS.2010.30`.
(Referenced on page 66.)

[LYH10] Ser-Hoon Lee, Yeo-Chan Yoon, and Sun-Young Hwang. Communication-aware task assignment algorithm for MPSoC using shared memory. *Journal of Systems Architecture*, 56(7):233–241, 2010. `http://dx.doi.org/10.1016/j.sysarc.2010.03.001`.
(Referenced on pages 66 and 67.)

[MAOM11] M. Mandelli, A. Amory, L. Ost, and F.G. Moraes. Multi-task dynamic mapping onto NoC-based MPSoCs. In *Proceedings of the 24th Symposium on Integrated Circuits and Systems Design*, pages 191–196. ACM, 2011.
(Referenced on pages 46 and 47.)

[MBQ02] Antoine Monsifrot, François Bodin, and René Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, AIMSA'02, pages 41–50, London, UK, 2002. Springer-Verlag. ISBN: 3-540-44127-1.
(Referenced on page 51.)

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976. `http://dx.doi.org/10.1109/TSE.1976.233837`.
(Referenced on pages 74, 75, and 76.)

[MDCE01] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–72, 2001.
(Referenced on page 26.)

[MF10] Andréa Matsunaga and José Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID'10, pages 495–504, Washington, DC, USA, May 2010. IEEE Computer So-

ciety. ISBN: 978-0-7695-4039-9. http://dx.doi.org/10.1109/CCGRID.2010.98.
(Referenced on pages 31, 38, 63, 64, 94, 156, and 177.)

[ML98] Philip Mucci and Kevin S. London. Low level architectural characterization benchmarks for parallel computers. Technical Report 394, UT Computer Science, July 1998. http://icl.cs.utk.edu/projects/llcbench/.
(Referenced on page 166.)

[MMK+11] Mohand-Said Mezmaz, Nouredine Melab, Yacine Kessaci, Young Choon Lee, El-Ghazali Talbi, Albert Y. Zomaya, and Daniel Tuyttens. A parallel bi-objective hybrid metaheuristic for energy-aware scheduling for cloud computing systems. *Journal of Parallel and Distributed Computing*, 71(11):1497–1508, 2011. http://dx.doi.org/10.1016/j.jpdc.2011.04.007.
(Referenced on pages 66 and 67.)

[MPI12] Message Passing Interface Forum. *MPI: A Message-passing Interface Standard, Version 3.0.* High-Performance Computing Center, September 2012. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report-book.pdf.
(Referenced on pages 41, 124, 130, 140, 147, and 163.)

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC'96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association. http://lmbench.sourceforge.net/.
(Referenced on page 166.)

[MTT+09] Damián A. Mallón, Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Andrés Gómez, Ramon Doallo, and José Carlos Mouriño. Performance evaluation of MPI, UPC and OpenMP on multicore architectures. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 5759:174–184, 2009. http://dx.doi.org/10.1007/978-3-642-03770-2_24.
(Referenced on page 42.)

[MU12] Stefan Metzlaff and Theo Ungerer. Impact of instruction cache and different instruction scratchpads on the WCET estimate. In *Proceedings of the IEEE 14th International Conference on High Performance Computing and Communication & IEEE 9th International Conference on Embedded Software and Systems*, HPCC-ICESS, pages 1442–1449, 2012. http://dx.doi.org/10.1109/HPCC.2012.211.
(Referenced on page 66.)

[MvWL+10] Matthias S. Müller, Matthijs van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C. Brantley, Chris Parrott, Tom Elken, Huiyu Feng,

and Carl Ponder. SPEC MPI2007 – an application benchmark suite for parallel systems using MPI. *Concurrency and Computation: Practice and Experience*, 22(2):191–205, 2010. `http://dx.doi.org/10.1002/cpe.1535`.
(Referenced on page 167.)

[NBvB+06] Florian Nigsch, Andreas Bender, Bernd van Buuren, Jos Tissen, Eduard Nigsch, and John B. O. Mitchell. Melting point prediction employing k-nearest neighbor algorithms and genetic parameter optimization. *Journal of Chemical Information and Modeling*, 46(6):2412–2422, 2006. `http://dx.doi.org/10.1021/ci060149f`.
(Referenced on page 34.)

[NC12] Nikita Nikitin and Jordi Cortadella. Static task mapping for tiled chip multiprocessors with multiple voltage islands. In *Proceedings of the 25th International Conference on Architecture of Computing Systems*, ARCS'12, pages 50–62, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN: 978-3-642-28292-8. `http://dx.doi.org/10.1007/978-3-642-28293-5_5`.
(Referenced on pages 66 and 67.)

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN: 3540654100.
(Referenced on page 123.)

[Nov04] Diego Novillo. From source to binary: The inner workings of GCC. *Red Hat Magazine*, 2, 2004. `www.redhat.com/magazine/002dec04/features/gcc`.
(Referenced on page 25.)

[NW82] Subhash C. Narula and John F. Wellington. The minimum sum of absolute errors regression: A state of the art survey. *International Statistical Review*, 50(3):317–326, 1982. `http://dx.doi.org/10.2307/1402501`.
(Referenced on page 33.)

[OB07] S. Arash Ostadzadeh and Koen L. M. Bertels. Parallelism utilization in embedded reconfigurable computing systems: A survey of recent trends. In *Proceedings of the 18th Annual Workshop on Circuits, Systems and Signal Processing*, ProRISC'07, November 2007.
(Referenced on pages 39 and 40.)

[OKS07] Ozcan Ozturk, Mahmut Kandemir, and Seung W. Son. An ILP based approach to reducing energy consumption in NoC based CMPs. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, ISLPED'07, pages 411–414, New York, NY, USA, 2007. ACM. ISBN: 978-1-59593-709-4. `http://dx.doi.org/10.1145/1283780.1283871`.

(Referenced on pages 66 and 67.)

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
(Referenced on pages 44 and 67.)

[OMP11] The OpenMP Architecture Review Board. OpenMP application program interface, version 3.1, July 2011. Up to date specifications are at `http://www.openmp.org`.
(Referenced on page 41.)

[OZSA07] Michael Ott, Jaroslaw Zola, Alexandros Stamatakis, and Srinivas Aluru. Large-scale maximum likelihood-based phylogenetic analysis on the IBM BlueGene/L. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC'07, pages 4:1–4:11, New York, NY, USA, 2007. ACM. ISBN: 978-1-59593-764-3. `http://dx.doi.org/10.1145/1362622.1362628`.
(Referenced on page 168.)

[OZW04] Márcio Seiji Oyamada, Felipe Zschornack, and Flávio Rech Wagner. Accurate software performance estimation using domain classification and neural networks. In *Proceedings of the 17th Symposium on Integrated Circuits and System Design*, SBCCI'04, pages 175–180, New York, NY, USA, September 2004. ACM. ISBN: 1-58113-947-0. `http://dx.doi.org/10.1145/1016568.1016617`.
(Referenced on pages 60 and 61.)

[PdSRdC13] Rattan Priya, Bruno Feres de Souza, André L.D. Rossi, and André C.P.L.F. de Carvalho. Predicting execution time of machine learning tasks for scheduling. *International Journal of Hybrid Intelligent Systems*, 10(1):23–32, 2013. `http://dx.doi.org/10.3233/HIS-130162`.
(Referenced on page 65.)

[PG08] Imran Patel and John R. Gilbert. An empirical study of the performance and productivity of two parallel programming models. In *22nd IEEE International Symposium on Parallel and Distributed Processing, Miami, Florida USA*, IPDPS'08, pages 1–7. IEEE, 2008. `http://dx.doi.org/10.1109/IPDPS.2008.4536192`.
(Referenced on page 41.)

[PIO13] Parallel I/O Benchmarking Consortium. *Parallel I/O Benchmarks*. Argonne National Laboratory, USA, 2013. `http://www.mcs.anl.gov/research/projects/pio-benchmark/`.
(Referenced on page 168.)

[PLL07] Saeed Parsa, Shahriar Lotfi, and Naser Lotfi. An evolutionary approach to task graph scheduling. In *Proceedings of the 8th International Conference on Adaptive and Natural Computing Algorithms, Part I*, ICANNGA'07, pages 110–119, Berlin, Heidelberg, 2007. Springer-Verlag.
(Referenced on page 67.)

[Qu01] Gang Qu. What is the limit of energy saving by dynamic voltage scaling? In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD'01, pages 560–563, Piscataway, NJ, USA, 2001. IEEE Press. ISBN: 0-7803-7249-2.
(Referenced on page 18.)

[Qui86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, March 1986. `http://dx.doi.org/10.1023/A:1022643204877`.
(Referenced on page 29.)

[R12] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. `http://www.r-project.org/`.
(Referenced on pages 148, 149, and 155.)

[RÅdSF13] Vítor Rodrigues, Benny Åkesson, Simão Patrício Melo de Sousa, and Mário Florido. A declarative compositional timing analysis for multicores using the latency-rate abstraction. Technical report CISTER-TR-130108, LIACC, Faculty of Computer Science, University of Porto, 2013.
(Referenced on page 66.)

[Ram00] Ganesan Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, March 2000. `http://dx.doi.org/10.1145/349214.349241`.
(Referenced on page 104.)

[RBAA04] Rodric M. Rabbah, Ian Bratt, Krste Asanovic, and Anant Agarwal. Versatility and versabench: A new metric and a benchmark suite for flexible architectures. Technical Report MIT-LCS-TM-646, Computer Science and Artifical Intelligence Laboratory, Massachusetts Institute of Technology Cambridge, MA 02139, June 2004.
(Referenced on page 166.)

[RC07] Rodric Rabbah and Nathan Clark. An infrastructure for research in backend compilation and architecture exploration, 2007. `http://www.trimaran.org/index.shtml`. CCCP Group at Georgia Institute of Technology and IBM Research.
(Referenced on page 166.)

[Rei96] Jeff Reilly. A brief introduction to the SPEC CPU95 benchmarks. *IEEE Computer Society TCCA Newsletter*, June 1996. `http://www.computer.org/tab/tcca/news/JUNE96/JUNE96_R.PS`.
(Referenced on page 166.)

[Rei07] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007. ISBN: 978-0-596-51480-8.
(Referenced on page 41.)

[RGB+06] Martino Ruggiero, Alessio Guerri, Davide Bertozzi, Francesco Poletti, and Michela Milano. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In *Proceedings of the 9th Conference on Design, Automation and Test in Europe*, DATE'06, pages 3–8, 3001 Leuven, Belgium, 2006. European Design and Automation Association. ISBN: 3-9810801-0-6.
(Referenced on pages 66 and 67.)

[RHJ09] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP'09, pages 427–436, February 2009. ISSN: 1066-6192. `http://dx.doi.org/10.1109/PDP.2009.43`.
(Referenced on pages 42 and 205.)

[Ric07] John A. Rice. *Mathematical Statistics and Data Analysis*, volume 72 of *Duxbury Advanced Series*. Duxbury Press, Belmont, CA, 3rd edition, 2007. ISBN: 9780534399429.
(Referenced on page 31.)

[RR10] Thomas Rauber and Gudula Rünger. Parallel programming models. In *Parallel Programming for Multicore and Cluster Systems*, pages 93–149. Springer Berlin Heidelberg, 2010. `http://dx.doi.org/10.1007/978-3-642-04818-0_3`. ISBN: 978-3-642-04818-0.
(Referenced on page 39.)

[RV13] Ciprian Radu and Lucian Vinţan. Domain-knowledge optimized simulated annealing for network-on-chip application mapping. In Loan Dumitrache, editor, *Advances in Intelligent Control Systems and Computer Science*, volume 187 of *Advances in Intelligent Systems and Computing*, pages 473–487. Springer Berlin Heidelberg, 2013. `http://dx.doi.org/10.1007/978-3-642-32548-9_34`. ISBN: 978-3-642-32547-2.
(Referenced on pages 66 and 67.)

[RW02] P. Rundberg and F. Warg. The freebench v1.0 benchmark suite, 2002. `http://www.elsniwiki.de/index.php/Main/FreeBench`.

(Referenced on page 166.)

[SA05] Mark Stephenson and Saman P. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the 3rd IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'05, pages 123–134. IEEE Computer Society, 2005. ISBN: 0-7695-2298-X.
(Referenced on page 51.)

[SBS13] Sandia Benchmarking Software. *Cbench: Scalable cluster benchmarking and testing*. Sandia National Laboratories, USA, 2013. `http://sourceforge.net/projects/cbench/`.
(Referenced on page 168.)

[SC13] Pradip Kumar Sahu and Santanu Chattopadhyay. A survey on application mapping strategies for network-on-chip design. *Journal of Systems Architecture*, 59(1):60–76, 2013. `http://dx.doi.org/10.1016/j.sysarc.2012.10.004`.
(Referenced on page 66.)

[SFT04] Warren Smith, Ian Foster, and Valerie Taylor. Predicting application run times with historical information. *Journal of Parallel and Distributed Computing*, 64(9):1007–1016, September 2004. `http://dx.doi.org/10.1016/j.jpdc.2004.06.008`.
(Referenced on page 61.)

[SIM+07] Karan Singh, Engin İpek, Sally A. McKee, Bronis R. de Supinski, Martin Schulz, and Rich Caruana. Predicting parallel application performance via machine learning approaches. *Concurrency and Computation: Practice and Experience*, 19(17):2219–2235, December 2007. `http://dx.doi.org/10.1002/cpe.v19:17`.
(Referenced on page 61.)

[SLM04] Alexandros Stamatakis, Thomas Ludwig, and Harald Meier. Parallel inference of a 10.000-taxon phylogeny with maximum likelihood. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Proceedings of the 10th International Conference on Parallel Processing, Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 997–1004, Pisa, Italy, 2004. Springer. ISBN: 3-540-22924-8. `http://dx.doi.org/10.1007/978-3-540-27866-5_134`.
(Referenced on page 168.)

[Smi00] Michael D. Smith. Overcoming the challenges to feedback-directed optimization (keynote talk). In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, DYNAMO'00, pages 1–11, New York, NY, USA, 2000. ACM. ISBN: 1-58113-241-7. `http://dx.doi.org/10.1145/351397.351408`.
(Referenced on pages 16 and 26.)

[Smi07] Ian Smith. Revised BYTE UNIX Benchmark Suite, 2007. http://code.google.com/p/byte-unixbench/.
(Referenced on page 166.)

[SNV+03] Dinesh C. Suresh, Walid A. Najjar, Frank Vahid, Jason R. Villarreal, and Greg Stitt. Profiling tools for hardware/software partitioning of embedded applications. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES'03, pages 189–198, New York, NY, USA, 2003. ACM. ISBN: 1-58113-647-1. http://dx.doi.org/10.1145/780732.780759.
(Referenced on pages 56 and 92.)

[Sta06] Alexandros Stamatakis. RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics*, 22(21):2688–2690, 2006. http://dx.doi.org/10.1093/bioinformatics/btl446.
(Referenced on page 168.)

[Ste09] Dan Steinberg. CART – classification and regression trees. In Xindong Wu and Vipin Kumar, editors, *The Top Ten Algorithms in Data Mining*, Chapman & Hall/CRC data mining and knowledge discovery series, chapter 10, pages 179–201. CRC Press, 2009.
(Referenced on pages 28 and 123.)

[SV08] Greg Stitt and Jason Villarreal. Recursion flattening. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, GLSVLSI '08, pages 131–134, New York, NY, USA, 2008. ACM. ISBN: 978-1-59593-999-9. http://dx.doi.org/10.1145/1366110.1366143.
(Referenced on page 56.)

[TJS+13] Shizhuo Tang, Naifeng Jing, Weiguang Sheng, Weifeng He, and Zhigang Mao. A thermal-aware task mapping algorithm for coarse grain reconfigurable computing system. In Weixia Xu, Liquan Xiao, Pingjing Lu, Jinwen Li, and Chengyi Zhang, editors, *Computer Engineering and Technology*, volume 337 of *Communications in Computer and Information Science*, pages 211–220. Springer Berlin Heidelberg, 2013. http://dx.doi.org/10.1007/978-3-642-35898-2_23. ISBN: 978-3-642-35897-5.
(Referenced on pages 66 and 67.)

[TKD12] Lalit Tembhare, Yogeshver Khandagre, and Aloke Dubey. Adaptive mapping for reducing power consumption on NoC-based MPSoC. *International Journal of Advanced Computer Research*, 2(4):471–474, December 2012.
(Referenced on pages 66 and 67.)

[Trä02] Jesper Larsson Träff. Implementing the MPI process topology mechanism. In *Proceedings of the 2002 ACM/IEEE Confer-*

ence on Supercomputing, Supercomputing'02, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. http://dx.doi.org/10.1109/SC.2002.10045.
(Referenced on page 128.)

[Tro10] John Tromp. The fhourstones benchmark, 2010. http://homepages.cwi.nl/~tromp/c4/fhour.html.
(Referenced on page 166.)

[TSYB11] Lothar Thiele, Lars Schor, Hoeseok Yang, and Iuliana Bacivarov. Thermal-aware system analysis and software synthesis for embedded multi-processors. In *Proceedings of the 48th Design Automation Conference*, DAC'11, pages 268–273, New York, NY, USA, 2011. ACM. ISBN: 978-1-4503-0636-2. http://dx.doi.org/10.1145/2024724.2024786.
(Referenced on pages 66 and 67.)

[Tur50] Alan M. Turing. COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 1950. http://dx.doi.org/10.1093/mind/LIX.236.433.
(Referenced on page 27.)

[UPC05] UPC Consortium. "UPC Language Specification, Version 1.2", Lawrence Berkeley National Lab Tech Report LBNL-59208, June 2005.
(Referenced on page 42.)

[Vap95] Vladimir N. Vapnik. *The nature of statistical learning theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1995. ISBN: 0-387-94559-8.
(Referenced on page 35.)

[VLCV01] Jason R.. Villarreal, R. Lysecky, S. Cotterell, and F. Vahid. A study on the loop behavior of embedded programs. Technical Report UCR–CSE–01–03, University of California, Riverside, 2001.
(Referenced on pages 56 and 92.)

[VR02] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Statistics and Computing. Springer, New York, fourth edition, 2002. http://www.stats.ox.ac.uk/pub/MASS4. ISBN 0-387-95457-0.
(Referenced on page 150.)

[Wan03] John Wang. *Data Mining: Opportunities and Challenges*. IGI Publishing, Hershey, PA, USA, 2003. ISBN: 1591400953.
(Referenced on page 28.)

[WB11] Miao Wang and François Bodin. Compiler-directed memory management for heterogeneous MPSoCs. *Journal of Systems Architecture*, 57(1):134–145, 2011. http://dx.doi.org/10.1016/j.sysarc.2010.10.008.
(Referenced on page 67.)

[WBD04] Youfeng Wu, M. Breternitz, and T. Devor. Continuous trip count profiling for loop optimization in two-phase dynamic binary translators. In *Eighth Workshop on Interaction between Compilers and Computer Architectures*, INTERACT-8, pages 3–12, February 2004. http://dx.doi.org/10.1109/INTERA.2004.1299505.
(Referenced on page 51.)

[WE12] Anthony Williams and Vicente J. Botet Escriba. Boost C++ Libraries: Threads. In *BoostBook*. (online), 2012. http://www.boost.org/doc/libs/1_53_0/doc/html/thread.html.
(Referenced on page 41.)

[WJM08] W. Wolf, A.A. Jerraya, and G. Martin. Multiprocessor system-on-chip (MPSoC) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, October 2008. http://dx.doi.org/10.1109/TCAD.2008.923415.
(Referenced on page 45.)

[WL94] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 1–11, New York, NY, USA, 1994. ACM Press. ISBN: 0-89791-707-3.
(Referenced on pages 24, 52, 57, 75, 76, 79, 114, 150, 151, 171, and 177.)

[WLQS13] Yi Wang, Duo Liu, Zhiwei Qin, and Zili Shao. Optimally removing intercore communication overhead for streaming applications on MPSoCs. *IEEE Transaction on Computers*, 62(2):336–350, 2013. http://dx.doi.org/10.1109/TC.2011.236.
(Referenced on pages 66 and 67.)

[WOT+95] S.C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, ISCA'95, pages 24–36, New York, NY, USA, June 1995. ACM. ISBN: 0-89791-698-0. http://dx.doi.org/10.1145/223982.223990.
(Referenced on page 166.)

[YH08] Hoeseok Yang and Soonhoi Ha. ILP based data parallel multi-task mapping/scheduling technique for MPSoC. In *Proceedings of the International SoC Design Conference*, volume 01 of *ISOCC'08*, pages –I–134–I–137, November 2008. http://dx.doi.org/10.1109/SOCDC.2008.4815591.
(Referenced on page 67.)

[YH09] H. Yang and S. Ha. Pipelined data parallel task mapping/scheduling technique for MPSoC. In *Proceedings of the 12th IEEE Conference on Design, Automation and Test in Europe*,

DATE'09, pages 69–74. IEEE, 2009. ISBN: 978-3-9810801-5-5. http://dl.acm.org/citation.cfm?id=1874620.1874638.
(Referenced on page 67.)

[Yoo09] Myungryun Yoo. Real-time task scheduling by multiobjective genetic algorithm. *Journal of System and Software*, 82(4):619–628, 2009. http://dx.doi.org/10.1016/j.jss.2008.08.039.
(Referenced on page 67.)

[YYWL13] Ziyu Yang, Ming Yan, Dawei Wang, and Sikun Li. A novel graph model for loop mapping on coarse-grained reconfigurable architectures. In Weixia Xu, Liquan Xiao, Pingjing Lu, Jinwen Li, and Chengyi Zhang, editors, *Computer Engineering and Technology*, volume 337 of *Communications in Computer and Information Science*, pages 231–241. Springer Berlin Heidelberg, 2013. http://dx.doi.org/10.1007/978-3-642-35898-2_25. ISBN: 978-3-642-35897-5.
(Referenced on page 67.)

[ZG13] Xibin Zhao and Ming Gu. A novel energy-aware multi-task dynamic mapping heuristic of NoC-based MPSoCs. *International Journal of Electronics*, 100(5):603–615, 2013. http://dx.doi.org/10.1080/00207217.2012.713179.
(Referenced on pages 66 and 67.)

[ZM69] G. Zyskind and F. Martin. On best linear estimation and general Gauss-Markov theorem in linear models with arbitrary nonnegative covariance structure. *SIAM Journal on Applied Mathematics*, 17(6):1190–1202, 1969. http://dx.doi.org/10.1137/0117110.
(Referenced on page 33.)

[Zuc96] David Zuckerman. On unapproximable versions of NP-complete problems. *SIAM Journal on Computing*, 25(6):1293–1304, December 1996. http://dx.doi.org/10.1137/S0097539794266407.
(Referenced on page 47.)

Publications by Dirk Tetzlaff

[TG10] Dirk Tetzlaff and Sabine Glesner. Intelligent task mapping using machine learning. In *Proceedings of the 2010 International Conference on Computational Intelligence and Software Engineering*, CiSE'10, pages 1–4. IEEE Computer Society, December 2010. ISBN: 978-1-4244-5392-4. http://dx.doi.org/10.1109/CISE.2010.5677019.
(Referenced on pages 15 and 90.)

[TG12a] Dirk Tetzlaff and Sabine Glesner. Making MPI intelligent. In Stefan Jähnichen, Bernhard Rumpe, and Holger Schlingloff, editors, *Proceedings of Software Engineering (Workshops)*, volume P-199 of *Lecture Notes in Informatics*, pages 75–88. GI, 2012. ISBN: 978-3-88579-293-2. http://subs.emis.de/LNI/Proceedings/Proceedings199/article6670.html.
(Referenced on pages 15 and 126.)

[TG12b] Dirk Tetzlaff and Sabine Glesner. Static prediction of recursion frequency using machine learning to enable hot spot optimizations. In *Proceedings of the 10th IEEE Symposium on Embedded Systems for Real-time Multimedia*, ESTIMedia'12, pages 42–51. IEEE Computer Society, 2012. ISBN: 978-1-4673-4968-0. http://dx.doi.org/10.1109/ESTIMedia.2012.6507027.
(Referenced on page 91.)

[TG13a] Dirk Tetzlaff and Sabine Glesner. Intelligent prediction of execution times. In *Proceedings of the Second International Conference on Informatics & Applications*, ICIA2013, pages 234–239. IEEE Computer Society, 2013. ISBN: 978-1-4673-5255-0. http://dx.doi.org/10.1109/ICoIA.2013.6650262.
(Referenced on pages 92 and 93.)

[TG13b] Dirk Tetzlaff and Sabine Glesner. Static prediction of loop iteration counts using machine learning to enable hot spot optimizations. In *Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications*, EUROMICRO-SEAA, pages 300–307. IEEE Computer Society, 2013. ISBN: 978-0-7695-5091-6. http://dx.doi.org/10.1109/SEAA.2013.12.
(Referenced on page 90.)

Talks by Dirk Tetzlaff

[Tet10] Dirk Tetzlaff. Intelligent task mapping for MPSoCs using machine learning, 2010. http://www.artist-embedded.org/docs/Events/2010/Map2MPSoC/slides/tetzlaff-map2mpsoc2010-wed-4.pdf.
Given talk at Map2MPSoC'10: 3rd Workshop on Mapping Applications to MPSoCs 2010.

[Tet11] Dirk Tetzlaff. Towards predicting recursion depth using machine learning, 2011. `http://www.artist-embedded.org/docs/Events/2011/Map2MPSoC/map2mpsoc-11-slides-tetzlaff.pdf`. Given talk at Map2MPSoC'11: 4th Workshop on Mapping Applications to MPSoCs 2011.

Supervised Theses

[Baa11] Abdenebi Baaddi. Intelligentes dynamisches Scheduling von MPI-Prozessen, 2011. TU Berlin, master's thesis, advisor: Prof. Dr. Sabine Glesner, further advisor: Dirk Tetzlaff.
(Referenced on page 160.)

[BT12] Franziska Bathelt-Tok. Statische Analyse des Laufzeitverhaltens von MPI-Prozessen, 2012. TU Berlin, bachelor's thesis, advisor: Prof. Dr. Sabine Glesner, further advisor: Dirk Tetzlaff.
(Referenced on page 158.)

[Ger13] Christian Gerson. Statische Analyse des Kommunikationsverhaltens von MPI-Prozessen, 2013. TU Berlin, bachelor's thesis, advisor: Prof. Dr. Sabine Glesner, further advisor: Dirk Tetzlaff.
(Referenced on page 159.)

[Gra11] Björn Gradowski. Effizientes Profiling von Funktionen, 2011. TU Berlin, bachelor's thesis, advisor: Prof. Dr. Sabine Glesner, further advisor: Dirk Tetzlaff.
(Referenced on page 151.)

[Roh11] Jens Rohnstock. Konzeption und prototypische Umsetzung eines serverbasierten Informationssystems für kritische Verkehrssituationen, 2011. TU Berlin, diploma thesis, advisor: Prof. Dr. Sabine Glesner, further advisor: Dirk Tetzlaff.

[See11] Volker Seeker. Design and implementation of an efficient instruction set simulator for an embedded multi-core architecture, 2011. TU Berlin / University of Edinburgh, diploma thesis, advisor: Prof. Dr. Sabine Glesner, further advisors: Björn Franke (CArD) and Dirk Tetzlaff (PES). This work was partially funded by the Network of Excellence on Embedded Systems Design ArtistDesign.

[Voi13] Andy Voigt. Vorhersage der Laufzeit von Funktionen mittels Maschinellen Lernens, 2013. TU Berlin, master's thesis, advisor: Prof. Dr. Sabine Glesner, further advisor: Dirk Tetzlaff.
(Referenced on page 154.)

List of Figures

List of Algorithms

List of Tables

List of Acronyms

ACET average-case execution time . . . 50
AI Abstract Interpretation . . . 53
ALU Arithmetic Logic Unit . . . 43
AMBA Advanced Microcontroller Bus Architecture . . . 45
API Application Programming Interface . . . 41
ASIP Application Specific Instruction-set Processor . . . 45
AST Abstract Syntax Tree . . . 22
BB basic block . . . 23
CCMIR Common CoSy Medium-level Intermediate Representation . . . 149
CFG Control Flow Graph . . . 23
CG Call Graph . . . 23
CoSy Compiler Development System . . . 147
CPU Central Processing Unit . . . 43
CU Control Unit . . . 43
DFA Data-flow Analysis . . . 25
DFG data flow graph . . . 59
DPRAM Dual-ported RAM . . . 45
DVFS Dynamic Voltage and Frequency Scaling . . . 18
FDO feedback directed optimization . . . 26
FPGA Field Programmable Gate Array . . . 44
FPU Floating-Point Unit . . . 43
GPU Graphics Processing Unit . . . 44
HIR High-Level IR . . . 23
HPC High Performance Computing . . . 147
ILP Integer Linear Programming . . . 24
IR Intermediate Representation . . . 23
ISA Instruction Set Architecture . . . 151
IWLS Iterated Reweighted Least Squares . . . 33
k-**NN** k-Nearest Neighbor . . . 34
LAD Least Absolute Deviations . . . 33
LIR Low-Level IR . . . 23
LSU Load/Store Unit . . . 43
MaCAPA framework for *Machine Learning based **Ma**pping of **C**oncurrent **A**pplications to **P**arallel **A**rchitectures* . . . 69
MIMD Multiple Instruction Multiple Data . . . 39