# MEMTRACE:
# A Memory, Performance and Energy Profiler Targeting RISC-Based Embedded Systems for Data-Intensive Applications

von
Diplom-Ingenieur
Heiko Hübert

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

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzende: Frau Prof. Dr. rer. nat. Sabine Glesner
Berichter: Herr Prof. Dr.-Ing. Hans-Ulrich Post
Berichter: Herr Prof. Dr.-Ing. Holger Blume

Tag der wissenschaftlichen Aussprache: 19. Mai 2009

Berlin 2009
D83

# Acknowledgements

The research described in this thesis has been carried out at the Fraunhofer Institute for Telecommunications, Heinrich Hertz Institute (HHI) in Berlin. My work within the Embedded System Group (ESG) of the Image Processing Department laid the foundations and provided inspiration for this dissertation.

First of all I would like to thank Professor Hans-Ulrich Post, who immediately agreed to supervise this thesis. He provided me with valuable comments on scientific and structural concerns. I am also very grateful to Professor Holger Blume for being the co-referee and for all our discussions, which were inspiring and valuable to me.

A very special and extended thank you goes to my supervisor at HHI, Dr. Benno Stabernack, who conducted me through all these years. I appreciate his advice on so many matters and the countless hours he invested in discussions and just chatting. I am also grateful to Dr. Ralf Schäfer, the head of the Image Processing Department, for giving me the opportunity to accomplish this work. My gratitude also goes to my colleagues at HHI for all the fruitful discussions and for helping me to expand my knowledge in many ways.

Special thanks go to Ella Ornstein, who helped this work to lose a bit of its German accent and make it nicer for the native speaker's ears. Also, Joshua Becker and my colleagues at HHI spent many hours of their spare time proof-reading. Many thanks.

Finally, I want to express my deepest appreciation to my family and friends. They provided moral support and always managed to encourage me whenever the goal seemed out of reach.

# Abstract

The design of embedded hardware/software systems is often subject to strict requirements concerning various aspects, including real time performance, energy consumption and die area. Especially for data-intensive applications, such as multimedia systems, the number of memory accesses is a dominant factor for these aspects. In order to meet the requirements and design a well-adapted system, it is necessary both to optimize the software and to design an adequate hardware architecture. For complex applications, this design space exploration can be difficult and requires in-depth analysis of the application and its implementation alternatives. This calls for profiling tools, which aid the designer in the design, optimization and scheduling of hardware and software.

Numerous tools exist for this purpose, and performance profiling solutions especially have been available for decades. Memory and energy profiling for embedded systems have become major issues within the last 10 years. However, the existing tools either cover only parts of the required profiling results or the statistics are not at the required level of detail. Some of the tools provide results only for the entire application and not at a source-code function level. This restricts the optimization potential, as the cause of a performance loss cannot be localized. Other tools suffer from a restricted level of accuracy. Results are based on generic processor architectures or taken with a low sample rate, or the tools apply source code instrumentation. Available profiling mechanisms with high accuracy suffer from long simulation times. This makes a comprehensive system analysis unfeasible.

This work presents a novel profiling methodology, which combines fast, accurate and comprehensive profiling in order to overcome the restrictions of the aforementioned techniques. The work describes the developed technique and its implementation as the MEMTRACE profiling tool. The trade-off between a decent simulation time and a sufficient level of accuracy is reached by using a tracing-based profiling approach that applies cycle-accurate simulators. In order to target a broad range of processors, a well-defined interface is established for interconnection with the processor simulator. Thus any cycle-accurate model can be used, as long as it provides access to basic runtime information such as the program counter, cycle counter and memory busses. The profiler is independent of the application's source code, which leads to higher accuracy as compared to instrumentation-based tools.

METRACE delivers cycle-accurate profiling results on a C function or even source code line level. The results include clock cycles, various memory access statistics and energy consumption estimates for embedded RISC-based processors. In addition to these results, the tool generates numerous statistics tailored to the specific optimization techniques that have been developed in this work. A focus is placed on memory access optimization, since for data-intensive applications, this aspect offers a high potential for increasing system efficiency.

Additionally to software analysis, the profiler supports an examination of bus-based systems, for example those composed of a processor, memory devices and coprocessors. For this purpose the coprocessors are represented by abstract but cycle-accurate models and MEMTRACE has been extended by detailed bus analysis features.

An instruction-accurate power consumption model has been developed for a sample processor and incorporated into the profiler for energy estimation. Two case studies are presented, which show how the applicability of the profiler and the optimization techniques has been proven in the design of hardware/software systems for data-intensive applications.

# Zusammenfassung

Der Entwurf eingebetteter Hardware/Software-Systeme unterliegt häufig strengen Anforderungen hinsichtlich verschiedener Kriterien wie z.B. Echtzeitfähigkeit, Energieverbrauch und Chipfläche. Insbesondere bei datenintensiven Anwendungen, beispielsweise in Multimediasystemen, spielt die Anzahl von Speicherzugriffen eine dominierende Rolle. Um diesen Kriterien beim Entwurf gerecht zu werden, muss sowohl die Software optimiert als auch eine adäquate Hardwarearchitektur entwickelt werden. Für komplexe Anwendungen kann diese Entwurfsraum-Exploration aufwändig sein und setzt eine detaillierte Analyse der Anwendung und ihrer Implementierungsalternativen voraus. Um den Entwickler bei Entwurf, Optimierung und Scheduling zu unterstützen, werden deshalb Analysewerkzeuge (Profiler) benötigt.

Zahlreiche Programme wurden bereits zu diesem Zweck entwickelt, insbesondere Leistungsanalysewerkzeuge existieren seit langem. Die Speicherzugriffs- und Verlustleistungsanalyse gewannen gerade in den letzten zehn Jahren an Relevanz. Die gegenwärtigen Profiler decken jedoch entweder nur einen Teil dieser Analysen ab oder sie können nicht den benötigten Detaillierungsgrad liefern. Beispielsweise können einige der Werkzeuge die Ergebnisse nicht den Quellcodefunktionen zuordnen. Dies verringert das Optimierungspotential, da die Ursache einer Leistungseinbuße nicht genau lokalisiert werden kann. Andere Profiler liefern hingegen eine eingeschränkte Genauigkeit aufgrund generischer Prozessormodelle, niedriger Abtastfrequenz oder Quellcodemodifikationen (Instrumentation). Analysemethoden mit hoher Genauigkeit benötigen oft lange Simulationszeiten, die eine umfassende Systemanalyse verhindern.

In dieser Arbeit wird eine neue Profilingmethode vorgestellt, die sowohl eine genaue, schnelle als auch umfangreiche Analyse ermöglicht und damit die Schwächen der erwähnten Methoden überwindet. Die Arbeit beschreibt die Methodik und deren Umsetzung als MEMTRACE Profiler. Durch einen tracingbasierten Ansatz, der einen zyklengenauen Simulator verwendet, können sowohl eine adäquate Simulationszeit als auch eine ausreichende Genauigkeit erreicht werden. Um einen breiten Bereich an Prozessoren abdecken zu können, wurde eine wohldefinierte Schnittstelle zwischen Profiler und Simulator geschaffen. Dadurch kann jedes zyklengenaue Modell verwendet werden, das einen Zugang zu grundlegenden Prozessorressourcen erlaubt, wie z.B. dem Befehlszähler, Zyklenzähler und den Speicherbussen. Außerdem ist der Profiler vom Quellcode der zu untersuchenden Applikation unabhängig, was zu einer höheren Genauigkeit gegenüber den Ergebnissen instrumentationsbasierter Ansätze führt.

MEMTRACE liefert zyklengenaue Analyseergebnisse auf Funktions- bzw. Zeilenebene des C-Quellcodes. Die Ergebnisse umfassen Taktzyklen, zahlreiche Speicherzugriffsstatistiken und Energieverbrauchsabschätzungen für eingebettete RISC Prozessoren. Neben diesen Ergebnissen werden zahlreiche weitere Analyseergebnisse generiert, die auf spezielle Optimierungen zugeschnitten sind, welche im Rahmen dieser Arbeit entwickelt wurden. Dabei wird ein Fokus auf Speicherzugriffe gelegt, da deren Optimierung bei datenintensiven Anwendungen ein hohes Potential zur Steigerung der Systemeffizienz mit sich bringt.

Zusätzlich zur Softwareanalyse wird durch den Profiler auch eine Untersuchung bus-basierter Systeme ermöglicht, z.B. bestehend aus einem Prozessor, Speichern und Coprozessoren. Dazu werden die Coprozessoren durch abstrakte, aber zyklengenaue Modelle abgebildet sowie MEMTRACE um detaillierte Busanalysefunktionen erweitert.

Um eine Abschätzung der Verlustleistung zu unterstützen, wurde exemplarisch ein instruktionsgenaues Verlustleistungsmodell entwickelt und in den Profiler integriert. Anhand zweier Fallstudien wird gezeigt, wie der Profiler erfolgreich innerhalb des Entwurfs von Hardware/Software Systemen für datenintensive Applikationen Anwendung finden konnte.

# Table of Contents

# Abbreviations

| | |
|---|---|
| AHB | Advanced High-performance Bus |
| ALU | Arithmetic Logic Unit |
| AMBA | Advanced Microcontroller Bus Architecture |
| API | Application Programming Interface |
| ASIC | Application Specific Integrated Circuit |
| ASIP | Application Specific Instruction Set Processor |
| AXF | ARM Executable Format |
| AXI | Advanced eXtensible Interface |
| CIF | Common Intermediate Format |
| CISC | Complex Instruction Set Computer |
| CLI | Command-line Interface |
| CMOS | Complementary Metal Oxide Semiconductor |
| CPU | Central Processing Unit |
| DDR | Double-Data-Rate |
| DLL | Dynamic Link Library |
| DMA | Direct Memory Access |
| DRAM | Dynamic Random Access Memory |
| DSP | Digital Signal Processor |
| DVB-H | Digital Video Broadcasting – Handheld |
| EBC | External Bus Cycles |
| ELF | Executable and Linking Format |
| FPGA | Field-Programmable Gate Arrays |
| FIFO | First In – First Out |
| FLI | Foreign Language Interface |
| FLPA | Functional-Level Power Analysis |
| GUI | Graphical User Interface |
| HDL | Hardware Description Language |
| ILPA | Instruction-Level Power Analysis |
| IO | Input/Output |
| IP | Intellectual Property |
| IPC | Instruction Per Cycle |
| ISO | International Organization for Standardization |
| ISA | Instruction Set Architecture |

| | |
|---|---|
| ISS | Instruction Set Simulator |
| ISSM | Instruction Set System Model |
| ITU | International Telecommunication Union |
| MMU | Memory Management Unit |
| MPEG | Moving Pictures Experts Group |
| NoC | Networks-on-Chip |
| OS | Operating System |
| PCB | Printed Circuit Board |
| PDA | Personal Digital Assistant |
| Pel | Picture Element (Pixel) |
| PLI | Programming Language Interface |
| QVGA | Quarter Video Graphics Array |
| RAM | Random Access Memory |
| RISC | Reduced Instruction Set Computer |
| ROM | Read Only Memory |
| RTL | Register Transfer Level |
| RTSM | Real Time System Model |
| RW | Read-Write |
| SIMD | Single Instruction Multiple Data |
| SPICE | Simulation Program with Integrated Circuit Emphasis |
| SoC | System-on-Chip |
| SDRAM | Synchronous Dynamic Random Access Memory |
| SI | International System of Units |
| SRAM | Static Random Access Memory |
| TCM | Tightly Coupled Memory |
| TLB | Translation Lookaside Buffer |
| VHDL | Very High Speed Integrated Circuit Hardware Description Language |
| VLC | Variable Length Coding |
| VLIW | Very Long Instruction Word |
| ZI | Zero Initialized |

# 1 Introduction

## 1.1 Background

The design of an embedded system often starts from a software description of the system in the C language. For example, the designer writes an executable specification based on a reference implementation of the application, e.g. from standardization organizations or the open-source community. This software code is often not optimized in any way, because it mainly serves the purpose of functional and conformance testing. Therefore it has to be transformed into an efficient system, including hardware and software components. The design of the system requires the following steps:

- system architecture design
- hardware/software partitioning
- software optimization
- design of hardware accelerators
- system scheduling

All these steps require detailed information about the performance of the different parts of the application. Besides the arithmetical demands of the application, memory accesses can have a huge influence on performance and power consumption. This is especially the case for data-intensive applications such as multimedia systems, due to the huge amount of data being transferred in these applications. This problem increases if the given data bandwidth is not used efficiently.

In order to reduce overall data traffic, those parts of the code which require a high amount of data transfer have to be identified and optimized. The above-mentioned applications contain up to 100,000 lines of source code. Therefore tools are required that help the designer to identify the critical parts of the software. Several analysis tools exist, for example gprof [42] or VTune [57] provide timing analysis. Memory access analysis is part of the ATOMIUM [25] tool suite. However, all these tools provide only approximate results for either timing or memory accesses. A highly accurate memory analysis can be done with a hardware (HDL) simulator, if an HDL model of the processor is available. However, such an analysis requires a long simulation time.

This thesis targets these issues and covers the performance, memory and power consumption profiling of embedded systems, as well as the usability of the profiling results within the design flow. In order to achieve a fast and accurate solution, a specialized profiler has been developed, called MEMTRACE, for obtaining performance, memory access and power consumption statistics. The profiling is tailored to embedded system architectures containing a single RISC processor, a single memory bus with memory-mapped components – such as memory or coprocessors – and a direct memory access (DMA) controller. This thesis will show how the provided profiling results can be used during the design and optimization of embedded hardware/software systems. Among other case studies, MEMTRACE has been applied during the efficient design of a mixed hardware/software system for H.264/AVC video decoding. Starting from a software implementation, this thesis shows how the software is optimized, an efficient hardware architecture developed and the system tasks scheduled based on the profiling results.

## 1.2 Contributions

The contributions of this work can be summarized as:

- presentation of an overview of existing profiling tools
- overview of embedded system architecture components
- design and implementation of a profiling tool suite
- development of several profiling analysis methods for memory-centric software and hardware optimization
- design and implementation of a hardware/software co-profiling environment
- creation of a simple processor power model for energy profiling
- integration of the profiler in an embedded system design flow
- application of the profiler for analysis of several software projects
- application of the profiler during the design of a system architecture for a multimedia SoC

This profiling and optimization methodology has been applied within several industrial and research projects. The profiling tool developed in this work has been used for evaluating and optimizing the performance of software targeting embedded devices. In-depth profiling has been performed, combined with system architecture exploration of memory and coprocessors.

The profiling methodology and the tool implementation have been presented to the research community in several publications and presentations at workshops and symposiums as well as in a book chapter [46, 47, 48, 92, 49]. Furthermore, the application of the tool within the design of embedded systems is described in technical journals and proceedings of international conferences [88, 89, 91, 87, 86]

## 1.3 Outline

Chapter 2 gives an overview of the state of the art in profiling and related subjects. A general overview of current embedded system design flows and existing tools for CPU, memory and power profiling is presented.

In Chapter 3 the theoretical background of the different aspects is surveyed. The components of processor-centric embedded systems and the corresponding simulation models are presented. Since the focus of this work is data-intensive applications, an example of such applications and implementation issues for embedded systems are given. The different aspects of profiling are highlighted and existing methods for hardware/software optimizations are presented.

Chapter 4 shows the contribution of this work to the field of profiling embedded processor systems. The profiling method and a variety of analysis results are presented. There is also a focus on power measurement and model creation for a sample processor. Furthermore, this thesis presents the application of the profiling results within the design flow for hardware- and software-centric optimizations.

The implementation and the workflow of the profiling tool as well as its integration with an existing instruction set simulator are described in Chapter 5. The profiling tool has been used within a number of projects for software analysis and optimization as well as for hardware architecture design. Some examples are presented in Chapter 6. Chapter 7 summarizes the work and points out unresolved issues and areas for future work.

# 2 State of the Art

This chapter presents the state of the art of profiling of embedded systems. As the term “embedded systems” is quite fuzzy, a review of the different definitions found in literature is provided and summarized. The major component of an embedded system is the processor, which controls the system and executes the software part of the application. An overview of the most common embedded processors is given. For profiling the software a simulation model of the processor is required, therefore Section 2.2 describes a processor simulator and a simulator generator. The following section gives an overview of the tools required in profiling process, including an example development toolchain and existing tools for performance, memory and power analysis.

## 2.1 Embedded Systems

Many definitions can be found for the term “Embedded System”. Marvedel gives a pragmatic definition in his book of “Embedded System Design” [66], which covers a wide range of application fields:

> *“Embedded systems can be defined as information processing systems embedded into enclosing products, such as cars, telecommunication or fabrications equipments”*

Most definitions slightly differ from each other, but a statement which can be found very often is that there is no fixed definition for this term. This fact was stated already a decade ago [39] and a fuzziness within the definition is still common today [66]. The most plausible reason is the enormous growth of the application field for embedded systems. Whereas an early application for embedded system could be found in banking transaction systems [100] and was implemented on mainframes. Over the decades their application field has grown to cover industrial control systems, networking devices, household appliances, automotive and consumer and many other products. Such systems can be found in portable as well as stationary devices. The following attributes can be found in most descriptions of embedded systems, e.g. in [66, 39, 72]:

- embedded in an enclosing device
- tailored to a specific application
- subject to real-time constraints and efficiency requirements
- interaction with external devices, such as sensors, regulators, input and output devices
- programmability is a common feature
- consists of hardware and software components
- often used in consumer products, control applications

Another way of identifying and defining embedded systems can be done by distinguishing the term from related devices or systems. ASICs, which are also tailored to a specific application, define devices usually composed of fixed hardware components. ASIPs come closer to embedded systems, or can be part of such, as they also offer programmability besides their restricted application area. However, ASIP only refer to the programmable part of a system. The definition of SoC comes closest to the one of embedded systems, as system-on-chips combine different components to form an entire system. An inherent characteristic of SoCs is that the parts are combined on a single chip, whereas embedded systems might be (and often are) constructed of multiple devices.

For clarifying the term, many people distinguish between embedded systems and general-purpose computer, such as a PC. The difference can be stated by a PC not being dedicated to a specific application and as such it can be programmed freely [21, 72]. On the other hand, the computing task of the embedded systems is invisible to the user [66]. Edwards et al. [32] state in 1997:

> *"Such systems, which use a computer to perform a specific function, but are neither used nor perceived as a computer, are generically known as embedded systems"*

The program code controlling the programmable parts of an embedded system is often stored in ROM, such as a flash memory and is usually referred to as firmware, contrary to the term software used in PCs.

Considering the two parts of the term, "embedded" indicates that the unit is part of a larger device and not stand-alone. And the word "system" reflects that it consists of several units, most often a processor, coprocessors and input/output units for interacting with the device.

### 2.1.1 Design Flow

The implementation of embedded hardware/software systems incorporates many design and optimization steps. The targeted application and requirements have to be mapped on a combination of hardware and software components. The mapping decision is influenced by several factors, mainly by the application requirements and the available hardware and software resources. For finding a suitable system architecture, usually a design space exploration is performed based on more or less detailed profiling [22].

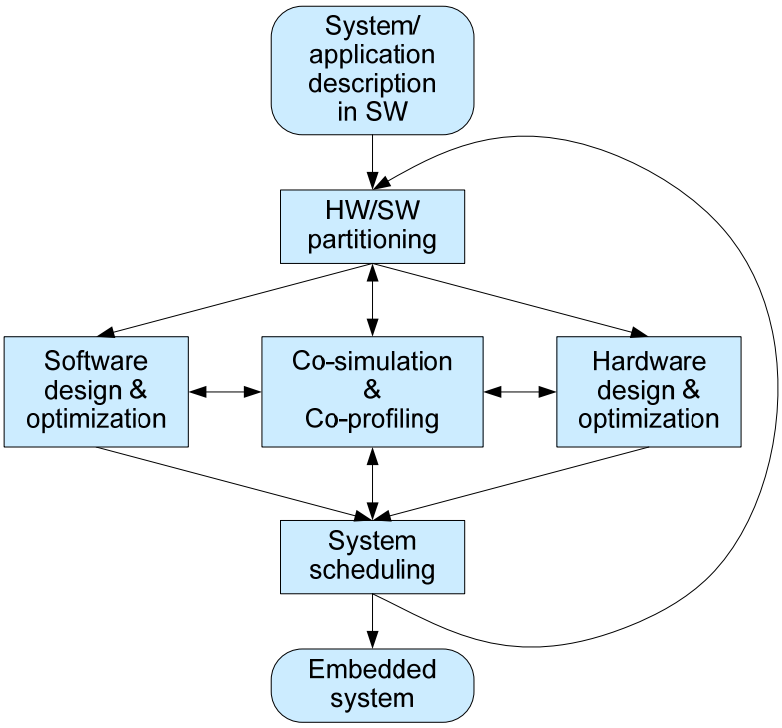



*Figure 1: Typical embedded system design flow*

Figure 1 shows a typical design flow for embedded hardware/software systems. Initially a system is defined in a textual form describing its functionality, requirements and constraints. The description of the functionality is then transferred to an executable form in order to prove its correctness. After functionally verifying this executable system description, often in C or

C++, a performance evaluation needs to be made in order to find an appropriate hardware/software architecture. Starting with a pure software implementation and an initial architecture specification, profiling can be used to measure the performance and reveal if the (real-time) requirements are met. If not, an iterative cycle of software and hardware partitioning, optimization and scheduling starts. During this process a continuous co-verification of the system is required. This includes on one hand a cosimulation [50] for ensuring the functional correctness of the system. On the other hand detailed profiling results are crucial for monitoring the influence of design steps on the performance.

### 2.1.2 Processors

Table 1 gives an overview of the most popular embedded RISC processors. Most processors feature a five- to seven-stage pipeline, as it offers a reasonable trade-off between maximum clock-frequency and instruction delay. A common number of registers is 32, with a major exception being the ARM processors, which only feature 16 registers. Section 2.1.3 describes the ARM processor family in more detail.

Most RISC processors are built as Harvard architecture with separated instruction and data caches. Typically, these caches are two- to four-way set associative and have a size of 32 to 64 kB. Since caches are very area and energy consuming, second-level caches are not very common in embedded system design and can only be found in high performance architectures, such as the ARM Cortex-A9 [8] SoCs. Some of the processors are extensible in their execution unit. The LEON [38] and the ARM processors use a coprocessor interface for this purpose. Before the calculation can be executed data needs to be transferred from the register file to the coprocessors by special instructions. Tensilica [94] and ARC [6] allow a customization of the instructions set. The execution unit can be placed beside the main ALU and incorporated in the pipeline path, which allows direct access to the register file. Besides the presented processors, numerous embedded architectures exist, which combine a RISC-Core with a DSP, e.g. Blackfin [5] from Analog Devices or TriCore [52] from Infineon. Such architectures have proven to be a good choice for combining control-flow and data processing needs of multimedia applications.

### 2.1.3 The ARM Architecture

The ARM processor architecture [37] has evolved over the years from a simple 3-stage pipelined RISC core to a 13-stage-pipelined multi-core SoC architecture. Initially, it had been developed only as a processor for personal computers, but the architecture has been found to be very efficient in terms of performance and power consumption. This makes the ARM processors a good candidate for embedded systems, e.g. for controlling tasks and the newer architecture types also for data processing. The ARM core architecture is a typical RISC processor however with a small register file of only 16 registers, which can lead to performance restrictions for data-intensive applications. Data transfer between register file and memory is only possible with load and store instructions. Newer core types also support more CISC-like load and store instructions for transferring multiple registers with a single instruction. These, instruction are often used for saving registers to the stack or for copying data from one memory location to another. Besides decreasing the instruction count it speeds up transfers by using the burst mode of the AHB.

*Table 1: Embedded RISC processors*

| Processor | Pipeline | Custom instr. | Registers | Instr./data cache (in bytes) | Instr./data TCM (in bytes) | Special features |
|---|---|---|---|---|---|---|
| **ARM7TDMI** | 3 | - | 16 | 8k unified | -/- | Von Neumann architecture |
| **ARM9E** | 5 | - | 16 | 128k/128k | +/+ | Coprocessor interface |
| **StrongARM SA-1110** | 5 | - | 16 | 16k/8k | -/512 | 32 set-associative caches, coprocessor interface |
| **XScale PXA27x** | 7-8 | - | 16 | 32k/32k | -/2k | SIMD, coprocessor interface, 256kB SRAM |
| **ARM11** | 8 | - | 16 | 64k/64k | +/+ | SIMD, branch prediction, 64-bit bus, coprocessor interface |
| **ARC600** | 5 | + | 32(-60) | 32k/32k | 512k/16k | Branch prediction, register file extendable |
| **ARC700** | 7 | + | 32(-60) | 64k/64k | 512k/256k | Branch prediction, 64-bit bus, register file extendable |
| **Tensilica Xtensa7** | 5 | + | >=64 | 32k/32k | 256k/256k | Up to 128-bit bus, windowed registers |
| **Tensilica Diamond232L** | 5 | - | 32 | 16k/16k | -/- | Windowed registers |
| **LatticeMicro32** | 6 | - | 32 | 32k/32k | -/- | |
| **Altera NIOS II** | 5-6 | + | 32 | 64k/64k | +/+ | Direct-mapped cache |
| **Xilinx MicroBlaze v5** | 5 | - | 32 | 64k/64k | +/+ | Direct-mapped cache, coprocessor interface |
| **MIPS 4KE** | 5 | - | 32 | 64k/64k | +/+ | Coprocessor interface |
| **openRISC OR1200** | 5 | + | 32 | 64k/64k | -/- | Direct-mapped cache, open source |
| **LEON3** | 7 | - | 520 | 1M/1M | +/+ | Coprocessor interface, windowed registers, open source |
| **SuperH SH-4/5** | 5 | - | 16 | yes/yes | -/- | 16-bit fixed length based, superscalar |

An exceptional feature of the ARM architecture is a barrel-shifter within the execution stage of the pipeline. The shifter is placed between the register file and the ALU. It can be used within data processing and also for extending the range of immediate value as well as for address offset manipulation.

Two major versions of the ARM instruction set exist, a full 32-bit version and a reduced 16-bit, also called Thumb instruction set. The 32-bit instructions support conditional execution, i.e. each instruction can be coded so that it only executes at a specific status of the CPU flags. The instruction set has grown with every new architecture version and has become very dense and irregular. Custom instructions are not allowed but custom functionality can be provided

via the coprocessor interface. Up to 15 coprocessors can be attached to a core where one coprocessor is already defined as system control coprocessor, e.g. for controlling the MMU and the caches. Specific instructions are available, which allow data transfer between coprocessor, external memory and core register file and furthermore for initiating data processing operations on the coprocessor. For most of the ARM processors support both, little and big endian data arrangement, thus depending on the application field the appropriate setup can be chosen.

ARM Ltd. licenses the processors as intellectual property and it comes in many different flavors. The oldest architecture available is the ARM7 family. These processors have a simple 3-stage pipeline and a von-Neumann architecture, i.e. using a shared data and instruction bus. The simplest version, the ARM7TDMI, only contains the core without a cache or MMU. This is a very small implementation of a RISC core with a low power implementation. Starting with the ARM9 family, a Harvard architecture is used, with separate caches for instructions and data. The pipeline is extended to five stages and an optional floating point unit is available. A write buffer is introduced for accelerating store operations to external memory and a memory management unit (MMU) for operating system support. The ARM9E family makes the caches customizable and adds tightly coupled memory (TCM) devices, which can be used for fast data and instruction access. The instruction set is extended with DSP instructions, such as a single-cycle multiply-accumulate instruction and saturating arithmetic. The processor used in most studies of this work, the ARM946E-S, is a member of this processor family.
The ARM 11 family, which was introduced in 2002, extends the pipeline to eight stages and splits it into a data execution and load/store pipeline. Branch prediction is used to decrease the need for flushes of the long pipeline. The instruction set is supplemented by SIMD instructions for use in data-intensive applications, e.g. video or data coding, and a built-in coprocessor for floating point arithmetic. Data transfers have been accelerated by a DMA controller for the TCMs, a wider memory interface of 64-bit and unaligned memory accesses.

The latest processor family is called Cortex and offers several core types targeting different application fields. The most powerful is the Cortex-A family, which provides a dual-issue 13-stage pipeline, extends the SIMD instruction set and width (128-bit) and adds a second-level cache. This family is not covered in this work, because a cycle-accurate instruction set simulator is not available within the software design suite (Real View Development Suite [14]).

Two further families based on the ARM architecture are the StrongARM [53] and the XScale [56] processors. In cooperation with Digital Equipment Corporation (DEC) ARM developed the StrongARM processors in 1995. This core is a predecessor of the ARM9 architecture and offers comparable hardware features, a 5-stage pipeline combined with a Harvard architecture with separate caches and MMUs. Remarkably, the caches, which had an initial size of 16 kB each (in later processor version the D-cache was reduced to 8 kB) are 32-way set-associative. From the programmers perspective it is more similar to the ARM7, the instructions set (ARMv4) provides only 32-bit instructions, without the Thumb or DSP extensions. When Intel took over the processor development from DEC, the StrongARM was improved and became the XScale processor. The XScale is compatible to the ARMv5TE instruction set, which is also supported by the ARM9E family. The pipeline is extended to 7-8 stages and the caches have a size of 32 kB each. Furthermore branch prediction is available and starting with the PXA270, the Wireless MMX extension provides SIMD instructions and an on-chip SRAM of 256 kB is incorporated.

Both, the StrongARM and the XScale are system-on-chip architectures, which provide power management features and numerous auxiliary components, such as DMA controller and inter-

faces to LC displays and serial data communication. They became very popular as processors for portable devices, e.g. PDAs or navigation systems.

### 2.1.4 The AMBA Architecture

Many system-on-chip architectures use busses that are compliant to the Advanced Microcontroller Bus Architecture (AMBA) [10] standard. AMBA is a royalty-free standard developed by ARM Ltd. and was first released in 1995. The standard defines a range of busses for different needs, starting from simple low-bandwidth busses for I/O purposes up to multi-channel pipelined busses for multi-core architectures. All AMBA busses are synchronous and have separate data and address busses.

The **Advanced Peripheral Bus (APB)** of the AMBA specification is optimized for low-bandwidth requirements, easy implementation and low power consumption and especially suitable for slow I/O components, such as timers and serial interfaces. The only master device allowed on the bus is the bridge to a higher order bus and the bus protocol is simple. Advanced features such as pipeline or burst transfer are not supported, and the bus width is restricted to 32 bit. In order to reduce the required chip area, a tristate implementation of the data bus is allowed. For faster system components such as the CPU, memory and DMA controller, the AMBA specification defines the **Advanced High-performance Bus (AHB)** architecture. The AHB interface is the standard bus connection for all ARM processors up to the ARM10 family, and is also the bus interface simulated in the ARMulator, and therefore used within this work. It is a multi-master compatible bus system and features separate read and write busses. An example system is shown in Figure 2, containing two bus masters and two slaves, the bus arbiter and the decoder.

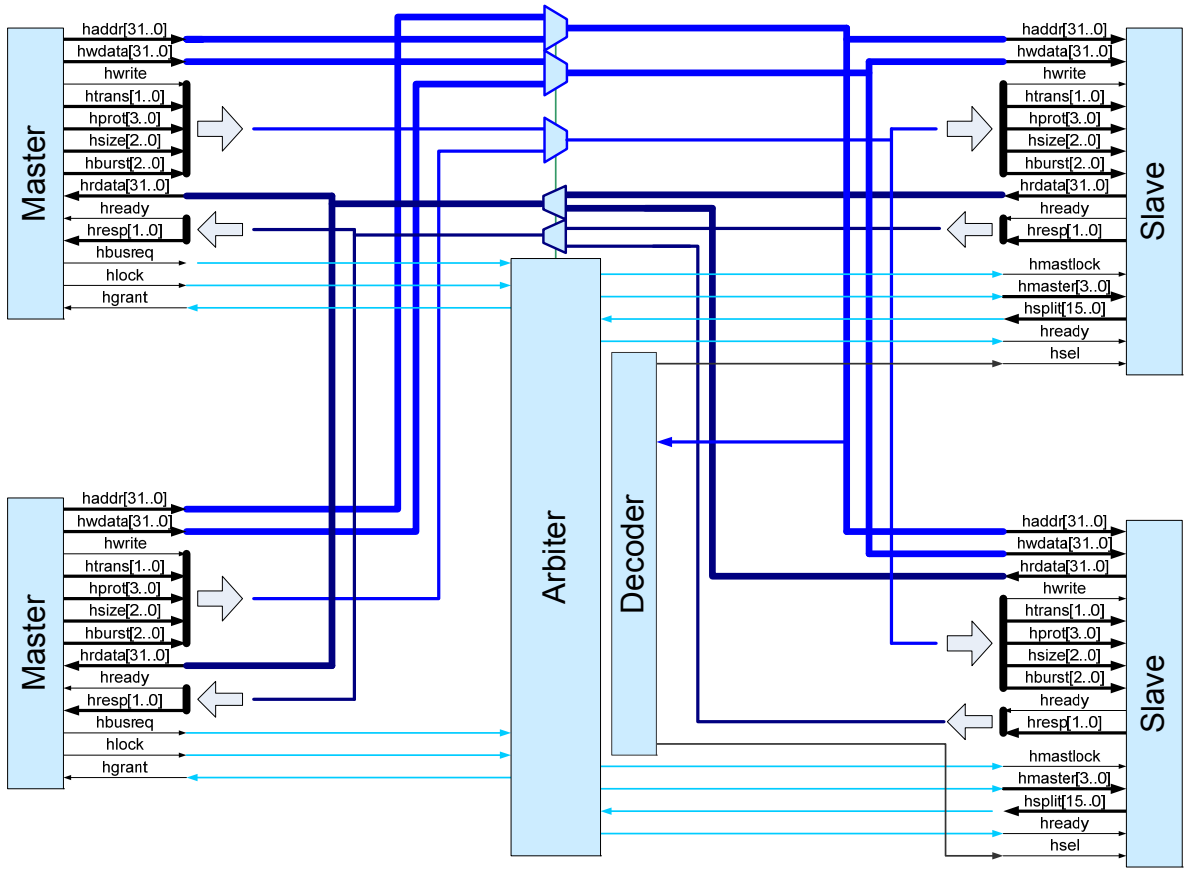



*Figure 2: AHB-based system with two masters and two slaves*

Any transaction is initiated by a master, which need to request access to the bus from the arbiter. According to a prioritization scheme, which is not specified in the standard, the arbiter grants access to the bus master with the highest priority. Once the access is granted, the arbiter sets the multiplexers giving the master access to all control, address and data busses, enabling it to reach the slaves. Although all slaves receive the signals, the actual addressed slave is selected by the decoder, which evaluates the address bus according to a specified memory map. In addition the multiplexers in charge of connecting the slave output signals, i.e. reading data and response signals, are set according to the memory map. The address and data signals are driven in a pipelined fashion, i.e. for one clock cycle the master provides the address and control signals, and in the next cycle the data values are expected on the bus. During the data phase of one access the master can issue the address for the next transaction. If the slave cannot serve the data signal in the next cycle, it prolongs the data phase by issuing a wait signal. If the slave expects the waiting time to be long, it can indicate this to arbiter, which may grant bus access to other masters, until the slave can serve the request. This so-called split transaction is especially useful when accessing slow-response slaves, such as an SDRAM controller during a page miss, in order to reduce idle time of the bus.

Besides single data transfer, the AHB protocol specifies burst transfer, which allows the transfer of multiple data values at consecutive addresses within one transaction. Similar to the single data transfer, it starts with an address phase, but features a multi-cycle data phase, where within each clock cycle (if not extended with wait states) one data transfer is issued. Burst transfers are very suitable for burst-oriented components, such as SDRAM, which require a long initialization (addressing) time for each access, but once addressed, can serve data very fast, in order to overcome a long delay time of single transfers.

The AHB protocol allows bursts of fixed sizes or undefined lengths. If another master is requesting the bus during a long burst operation, this leads to a long latency time for this master, and may reduce the system speed significantly. Therefore, a feature called early burst termination is available, which allows the arbiter to decide to interrupt the burst and grant the bus to other waiting masters. For performance increase of the AHB, the simple multiplexer structure shown in Figure 2 can be replaced by a more complex interconnect matrix, which creates a multi-layered bus architecture for parallel data transfer.

The new bus standard within the AMBA specification is the **Advanced eXtensible Interface (AXI)** Protocol [11]. It targets even higher bandwidth requirements, especially by separating address and data busses and allow multiple data busses to exist in the system. Thus, multiple transactions can be issued on the address bus and served independently on the data busses. Furthermore, separate read and write data and address busses are defined, which allows concurrent read and write transactions. Each transaction can be labeled with a transfer ID for allowing out-of-order completion. If some transactions have higher latency responses than others, the out-of-order handling leads to less idle time on the busses. The AXI-based busses are the standard interface of never ARM processors and are used in ARM11 and Cortex cores.

## 2.2 Processor Simulators

For testing and analyzing software for a specific processor, the processor needs to be available as a hardware device or simulator, whereas simulators usually offer a more elaborate view of the processor internals. Simulators exist for most processors, differing in their accuracy, sometimes even multiple models on different accuracy levels exist for a processor [81]. As an example of a typical model, an instruction set simulator (ISS) for the ARM architecture is described in more detail. Usually processors, as any digital hardware component, are de-

scribed in a hardware description language. If no other processor model on a higher abstraction level is available, these models can be used for simulation. Being very fine grained they lead to long simulation times. In Section 2.2.2 a tool is presented, allowing a transformation of such models to a higher level of abstraction for faster simulation.

### 2.2.1 ARMulator – The ARM Instruction Set Simulator

The ARMulator [15] is the ISS for processors based on the ARM architecture. In conjunction with a debugger it can be used for code evaluation and performance analysis. Besides the processor core including the pipeline and the register file, the ARMulator simulates other architectural features such as caches, a memory management unit (MMU) and a memory sub system and peripheral devices. The ARMulator is implemented as DLL and works together with the RealView Debugger, the AXF debugger and the command-line debugger armsd, which are all part of the RealView Development Suite, see Section 2.3.1.

The ARMulator simulates the processors in a nearly cycle-accurate manner. Restrictions on the accuracy apply concerning the cache simulation and advanced memory bus (AHB) architectures. The simulator supports a wide range of processors based on the ARM architecture, including a basic support for StrongARM and XScale processors from Intel. See Section 2.1.3 for more details on the ARM processor architecture. As the ARM cores are available as hardware IPs, many features of the cores are adjustable. The simulator allows adjusting these features, which include:

- cache sizes and organization
- tightly coupled memory size
- processor speed
- divider between processor and bus cycle length

The external memory bus architecture is an abstract model of the AMBA AHB standard, which defines a multiplexer-based on-chip bus. The multi-master capable bus has separate address and data busses and a typical data bus width of 32-bit. The simulator can be extended by modules, which can be used for gathering inside information of the simulator or for the simulation of hardware components surrounding the processor. Such hardware extensions can either be memory-mapped devices or connected via the coprocessor interface, thus creating an entire system architecture. Some examples for extension modules are already provided with the ARMulator, such as a profiler and a tracer module for analysis purposes as well as memory module for simulation of a memory sub system. Figure 3 shows an example connection of modules to the ARMulator.

The ARMulator is extended with the tracer module, tracing all accesses to the caches and from the caches to the external bus. The Mapfile models the timing behavior of each address region defined by a memory map file. Every bus access is then passed to a bus model (Flatmem), which performs the address decoding, and parses the access either to a simple memory model or to memory-mapped peripherals, such as a timer component or an interrupt controller.

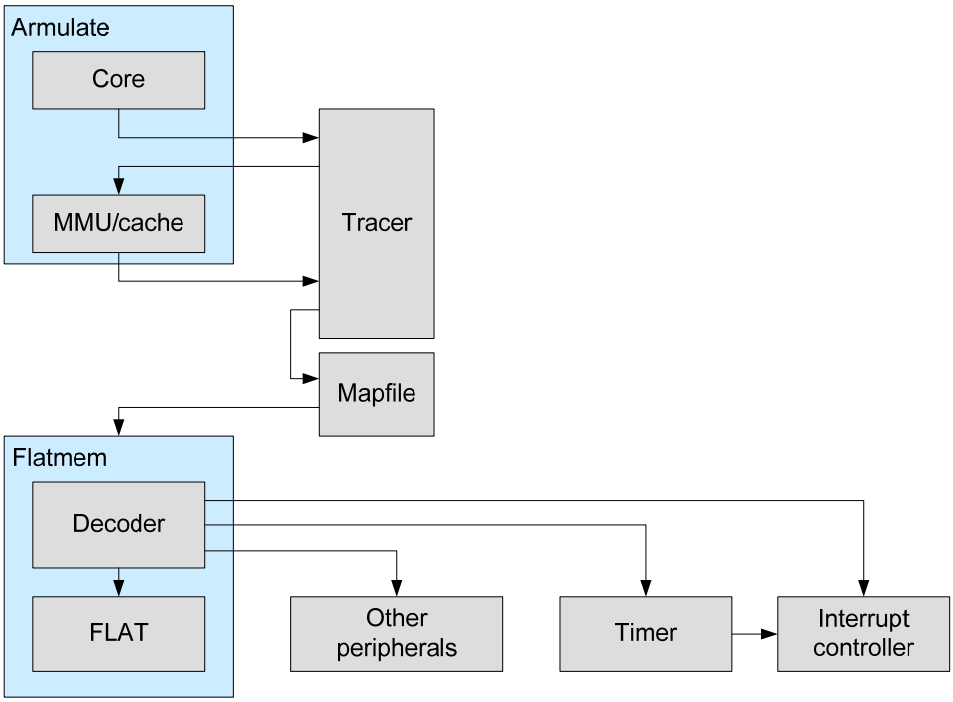



*Figure 3: ARMulator extended with modules [15]*

Modules need to be written in C or C++ and are connected to the ARMulator as DLLs. An API is defined for the interconnection and the ARMulator extension kit provides the required header files, libraries and makefiles. The API consists of numerous functions to access inside variables of the simulator. This includes:

- read or write of a register or coprocessor value
- assert or read signals, e.g. interrupt signal
- set or get events, e.g. from the CPU/MMU or other modules, such as cache miss or address undefined instruction
- access any memory location (without interfering the actual bus simulation)
- control and access to the simulator internals, such as reading the cycle counter or adding other counters
- accessing the debugger, e.g. for printing messages on the debugger screen

The API also defines a number of auxiliary functions, helpful during the design of modules. For example, for every instruction set a disassembly API function exists, providing the disassembled line for a binary instruction code word. For actively including modules into the simulation process, callback functions can be registered in the simulator. The functions can be called at any of the following occasion:

- each instruction: for instruction tracing
- bus cycle: for inspecting bus behavior
- event: with installing event handler, this event can be caught within a module and perform a particular action in the module
- after a specific time (from now): e.g. to imitate delay behavior of real hardware

Bus modules can be considered as leafs of the memory architecture tree generated by the Flatmem module. They are called as soon as a memory access to their address range occurs.

The address range needs to be registered by the API function `ARMulif_ReadBusRange()` and the module function, which serves the memory access within the module is registered with the API function `bus_registerPeripFunc()`. Section 2.2.1.2 describes the definition of a generic bus module for memory-mapped devices.

Memory modules, such as the tracer module, are instanced by being linked into the memory chain. Starting from the core, every memory access is passed from one link in the memory chain to the next. Two memory busses are accessible to modules, the core memory bus and the external memory bus. The former is the connection between the core and the caches. The latter connects the caches with the memory subsystem if caches are available, otherwise memory and core bus are identical. Memory modules can link to both busses. The API functions `ARMulif_QueryMemInterface()` provides a handle to a bus, which is required for connecting to the bus and for retrieving information about the bus type. The function `ARMul_InsertMemInterface()` is then applied to insert the module into the memory chain and provided the simulator with the appropriate callback functions. The following section about the Tracer module gives an example of a memory module.

The ARMulator, as well as the modules, can be controlled by configuration files. These files specify module parameters or allow disabling a module, which might be required for speeding-up the simulation. Alternatively these parameters can be overwritten with values passed from the debugger, e.g. with command-line parameters of the debugger in order to modify the behavior of a module for the current simulation run.

For new processors, starting with the Cortex family, the ARMulator has been replaced with the Instruction Set System Models (ISSMs) [8], which no longer provide cycle-accuracy. Instead, in 2007 ARM presented the SOC Designer tool suite, which provides cycle-accurate processor and system models. In 2008 ARM discontinued the SOC Designer development and the tool was acquired by Carbon Design Systems. Thereafter ARM focuses on hardware-based profiling instead, which is supported by ARM RealView Profiler, see Section 2.3.2.3.

#### 2.2.1.1 The Tracer Module

The Tracer module is an extension for tracing numerous processor activities, such as instructions, memory accesses, register changes, and events, such as cache misses, and writes them to file. An example trace file is given in Listing 1.

```
MSW4____ 0001A190 23C06023
BNR4O___ 000080C0 28B00030
MNR4O___ 000080C0 28B00030
IT______ 000080B8 8affffb BHI      0x80ac
R_______ r1=0001a194
BNR4O___ 000080AC E2522010
MNR4O___ 000080AC E2522010
BSR4O___ 000080B0 28B01070
MSR4O___ 000080B0 28B01070
BSR4O___ 000080B4 28A11070
MSR4O___ 000080B4 28A11070
IT______ 000080AC e2522010 SUBS     r2,r2,#0x10
BSR4O___ 000080B8 8AFFFFFB
MSR4O___ 000080B8 8AFFFFFB
IT______ 000080B0 28b01070 LDMCSIA  r0!,{r4-r6,r12}
R_______ r2=00000018
BNR4____ 00019FA0 C023C184
```

*Listing 1: Example tracer file*

The first letter of each line indicates the information type; the “M” stands for memory access on the core memory bus, “B” lines indicate an access on the external memory bus. The accessed address and data are supplied. Lines starting with an “I” indicate the executed instruction including the disassembly, the suffix “T” indicates that the instruction was taken. “R” shows changes in registers. Usually, for more complex software millions of cycles need to simulated, thus the trace files can become enormously large, in the range of gigabytes. Therefore, tracing a whole software execution with full trace information is not feasible. The tracer allows disabling specific trace information, limiting the address range for memory tracing and sub-sampling the tracing, i.e. only every n-th tracing sample will be written to the trace file. The trace file source code is part of the RealView environment.

The tracer module connects to the ARMulator with four interfaces. It installs itself as bus module on the core bus, for tracing every access to the instruction and the data cache. Furthermore it links into the external bus chain, for tracing cache and write buffer accesses to the external memory and other bus components, as shown in Figure 3. Instruction usage is analyzed by installing a so-called hourglass callback function, which is called each time a new instruction is decoded. The disassembly functions are used to produce the disassembly string depending on the current instruction mode. And finally, a callback function tracing various events is installed.

#### 2.2.1.2 The Mapfile Module

The Mapfile module defines a timing behavior for bus devices. The timing is provided in wait states in terms of bus clock cycles. The Mapfile differentiates between read and write access times as well as between sequential and non-sequential accesses. The specific timing for sequential accesses can be used to emulate for example burst modes on the bus or page hits in DRAMs. Although this timing does not reflect the real behavior, at least it allows an approximation. For more detailed timing a DRAM timing module would be required. The Mapfile imitates the AMBA bus behavior in a simple manner.

```
;; start     size  name width access read-times write-times
00000000 00010000   ROM     2      r        8/8         0/0
00010000 000F0000 NOMEM     4      -        0/0         0/0
00100000 00100000  SRAM     4     rw      16/16       16/16
00200000 00800000  DRAM     4     rw      208/8       200/8
00A00000 FF600000 NOMEM     4      -        0/0         0/0
```

*Listing 2: Timing definition for the Mapfile module*

The timing parameters are defined in a file, a typical example of an embedded system architecture is given in Listing 2. The mapping defines a ROM with a size of 16 kB (0x10000 bytes) starting at address 0x0. The data width of the ROM is 16 bits (2 bytes) and the sequential and non-sequential read access times are 8 ns each. An SRAM resides at address 0x10000 with a size of 1 MB and read and write times of 16 ns.

The non-sequential access times (208 ns and 200 ns) of the DRAM component are much slower than the sequential times. This reflects the page architecture of DRAMs [45]. DRAMs are organized in pages of memory cells, which are activated at the same time. Pages have a size of typically 0.5 to 4 kB, and accesses within a page are served very fast. An access to another page results in a page miss. For accessing this new page a pre-charge is required, leading to a longer access time. Furthermore, if sequential accesses within a page occur, a burst mode can be used, decreasing the access time even further. This results in the short access times (8 ns) for sequential accesses to the DRAM.

Depending on memory bus speed the Mapfile calculates the resulting wait states. For example the 208 ns for a non-sequential read access to the DRAM lead to 25 wait states on a bus running at 125 MHz. Every access to a memory location from the processor simulator passes through the Mapfile module. The Mapfile evaluates the timing behavior for this access, and either passes the access to the memory, if no wait states occur, or triggers the wait signal on the bus.

### 2.2.2 Automatic Simulator Generation – The Verilator

If simulators are not available for an existing processor or if the processor is a self-developed design, a model of the processor needs to be created. One choice is writing a model of the processor in software, for example in C. It is tricky to guarantee the consistency of the real hardware processor and the software model, as they are developed independently. This task becomes even more complicated if the processor design changes during development, for example the pipeline structure is rearranged or new instructions are added. One solution is to use the hardware description of the processor, if available, which is usually written in a hardware description language, such as Verilog or VHDL. Simulation tools exist for executing the hardware model, and a connection to the profiler can be established via specific external interfaces, for example via the so-called foreign or programming language interface (FLI, PLI). However, hardware simulation is usually performed on a nanosecond-accurate level. This implies long simulation time and delivers an accuracy, which is not required for the profiling. Another choice is to convert the HDL model to a faster and more abstract cycle-accurate model. Different tools exists to convert HDL models to SystemC or C, e.g. V2SC [68], Verilator [85], or VHDL-to-SystemC [98]. Within this work the Verilator tool is applied for automatic generation of a processor model. The tool takes a set of synthesizable Verilog code files and creates a C++ model of this code including a simulator environment. The result is a C++ class containing the top level module of the Verilog design and all lower level modules. The input and output ports of the top level module are converted to variables with the same name. In addition ports and signals of lower level modules are still visible.

The Verilator environment allows an easy testbench creation for the generated processor. Listing 3 shows the testbench for a C++ processor model called “Vtop”. Similar to the HDL model the processor is operated with the reset signal and a toggling clock signal. The simulator is instructed to evaluate the input signal and generate the internal and output signals by calling the eval() function. Furthermore a tracing module is provided, which can be used to generate waveform of signals.

```
#include <verilated.h>      // Defines common routines
#include "Vtop.h"           // From Verilating "top.v"
Vtop *top;                  // Instantiation of module
...
int main() {
  top = new Vtop;  // Create instance
  top->reset_l = 0;// Set some inputs
  while (!Verilated::gotFinish()) {
    if (main_time > 10) {
      top->reset_l = 1; // Deassert reset
    }
    if ((main_time % 10) == 1) {
      top->clk = 1; // Toggle clock
    }
    if ((main_time % 10) == 6) {
      top->clk = 0;
```

```
    }
    top->eval(); // Evaluate model
    cout << top->out << endl; // Read a output
    main_time++; // Time passes...
  }
  top->final(); // Done simulating
}
```

*Listing 3: Example code of a C++ testbench for a Verilator processor simulator [85]*

## 2.3 Tools

This section presents tools for the design and profiling of embedded software. As an example of a software development suite, the RealView Developer Suite [14] for ARM processors is presented. It contains tools for compiling and building applications from source or assembly code targeted to a specific processor architecture. Development suites for other processors provide similar features and tools, for example the GNU Compiler Collection [36]. In the following sections a survey of existing profiling tools is given, which cover the analysis of performance, memory accesses and power consumption.

### 2.3.1 ARM Software Development Toolchain

ARM provides a toolchain for software development targeted to their processors, which is called RealView Developer Suite [14]. The development suite includes all tools required for software development, including compiler, linker, debugger and ISS.

Armcc [13] and armasm are the compiler and assembler for creating object files from software source code files. The tools support the compilation of ISO C, ISO C++ or ARM assembly code, respectively and are equipped with the standard library sets for C and C++ including file input/output and the Standard Template Library (STL). The code generation of the tools can be controlled with command-line options for creating code optimized for specific processor types and instruction sets, floating-point implementations and different optimizations targets, i.e. small code size or fast execution. Floating-Point operations can be either coded as assembly instructions or as calls to library functions. The former requires that the processor is equipped with a floating-point unit or an emulator. The tools can be instructed to include debug information, e.g. for use in a debugger, in order to map assembly code lines to C source code lines.

The object files created by armcc or armasm can either be first combined to a library with the armar achiever tool, or directly linked to an executable file with the linker utility (armlink). The linker creates an executable files from object and library files in the ARM Executable and Linking Format (ARM ELF). The linker defines the placement of the code and data segments of the input files in memory. Normally, code and data are partitioned in three regions:

- ER_RO: read-only region for program code and constant data
- ER_RW: read-write region for global variables
- ER_ZI: zero-initialized region for data, which need to initialized with a zero value

The regions are usually placed consecutively in memory followed by the heap. As stack and heap grow toward each other, the stack is usually placed at a high address, in order to avoid a stack-heap collision. The placement of the regions can be controlled by command-line options of the linker, e.g. the start address and the grouping of regions. For a more comprehensive

definition of the memory map a so-called scatter-loading can be applied. Within a scatter file, the exact placement can be specified, see Listing 4.

```
ROM_LOAD 0x0 {
    ROM_EXEC 0x0 {
        vectors.o (Vect, +First)
        * (+RO)
       }
    DRAM_RW +0 {
        .ANY (+RW)
    }
    DRAM_ZI +0 {
        .ANY (+ZI)
    }
    HEAP +0 UNINIT {
        heap.o (+ZI)
    }
    TCM 0x04000000 0x2F40 {
        tcm_vars.o (+RO,+RW,+ZI)
    }
    STACKS 0x28080000 UNINIT {
        stack.o (+ZI)
    }
}
```

*Listing 4: Example scatter file*

The interrupt vectors and the startup code, defined in the file vectors.o are placed in at address 0x0 followed by the RW and ZI region and the heap. A number of variables, defined in tcm_vars.o, are placed in a fast tightly coupled memory (TCM), which is memory-mapped at location 0x04000000 with a size of 0x2F40. Finally, the stack is placed at address 0x28080000.

The executable files created by the linker can be preprocesses and examined by any ELF compatible tool. The RealView suite includes a tool called fromelf, which can be used for converting the executable into another format, e.g. binary format. Additionally it displays the content of an ELF file, including code disassembly, debug information, and symbol tables. In Section 4.1.1 it is shown, how the tool can be used for extracting function and variable names of the executable.

The tool suite provides two GUI-based debuggers and a command-line-based debugger called armsd for evaluating and debugging the code. The debuggers can either be used for observing code execution on real processor hardware or on a simulator. The connection to the execution unit is established via a DLL interface. For simulation the debugger connect to the ARMulator, which is described in Section 2.2.1 The ARMulator can be extended for simulating the behavior of surrounding memory-mapped hardware, e.g. memory, coprocessors or DMA controllers.

### 2.3.2 Cycle Profiling Tools

Profiling is a part of the software development process and describes the procedure of analyzing the execution behavior of software concerning different metrics. Table 2 gives an overview of existing profiling tools. They differ in the delivered analysis results, accuracy and supported processor architectures. The tools are explained in more detail in the following sections.

*Table 2: Profiling tools*

| | Cycles | Memory accesses | Power | Per function | Per line | Callgraph results | Instrumentation/ source code required | Accuracy | Embedded processors |
|---|---|---|---|---|---|---|---|---|---|
| **Gprof** | + | - | - | + | + | + | + | Sampling (10 ms) | |
| **Armprof** | + | - | - | + | - | + | - | | ARM CPUs |
| **ARM RealView Profiler** | + | + | - | + | + | + | - | Sampling (µs/ns) & estimation | ARM CPUs |
| **ATOMIUM** | + | + | - | + | - | + | + | | Abstract model |
| **PowerEscape**[1] | + | + | + | + | - | + | + | | Abstract model |
| **VTune** | + | + | - | + | + | +[2] | - | Sampling | Xscale, Intel x86 |
| **HDL Profiling**[3] | + | + | + | - | - | - | - | Ns | If HDL model available |
| **Valgrind / Callgrind** | + | + | - | + | + | + | - | Simulated CPU | Only x86/PPC |
| **SimpleScalar** | + | + | + | - | +[4] | - | - | Simulated CPU | Synthetic model |
| **MEMTRACE** | + | + | + | + | + | - | - | Cycle | if ISS available |

[1]no longer available
[2]not for embedded processors
[3]very slow
[4]per assembly address

#### 2.3.2.1 Gprof

Gprof [42] is a callgraph profiler, which was developed in the early 1980's at the University of Berkeley. It is based on the UNIX profiling tool prof [19] and became part of the BSD-UNIX system. With some enhancements gprof also became part of the binutils package [35] and is therefore available on all GNU/Linux systems. Thus, gprof became a widely-used profiling tool for software analysis and optimization.

Gprof is based on instrumentation of the source code. This process is performed by the compiler and needs to be enabled by designer manually, i.e. the gcc compiler provides the "-pg" option for this purpose. During instrumentation, see Section 3.3.2.1 for more details, the source code is enhanced with code fragments, which are responsible for generating profiling data during execution of the program. After finishing the program execution, the collected profiling information is written to a file, which can be further processed by the profiler. Gprof basically provides two different profiles, a flat profile as depicted in Listing 5 and a callgraph profiling shown in Listing 6.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 65.84     15.60    15.60    20000   780.22   883.96  b
```

```
 17.51      19.75      4.15     40000    103.74    103.74  c
 17.13      23.81      4.06     20000    202.97    306.71  a
```

*Listing 5: Gprof flat profile*

The flat profile is similar to the results that the simple prof tool delivers, including the total time spent in a function, the number of calls, and the average time per call. An enhancement is the total time per call including the called functions. This information is based on the callgraph profiling.

```
granularity: each sample hit covers 2 byte(s) for 0.04% of 23.81
seconds

index % time    self  children    called     name

[1]    100.0    0.00   23.81                 main [1]
               15.60    2.07   20000/20000       b [2]
                4.06    2.07   20000/20000       a [3]
-----------------------------------------------
               15.60    2.07   20000/20000       main [1]
[2]     74.2   15.60    2.07   20000         b [2]
                2.07    0.00   20000/40000       c [4]
-----------------------------------------------
                4.06    2.07   20000/20000       main [1]
[3]     25.8    4.06    2.07   20000         a [3]
                2.07    0.00   20000/40000       c [4]
-----------------------------------------------
                2.07    0.00   20000/40000       a [3]
                2.07    0.00   20000/40000       b [2]
[4]     17.4    4.15    0.00   40000         c [4]
```

*Listing 6: Gprof callgraph profile*

The detailed results for the callgraph profile provide a section for each node in the callgraph, see Figure 4 for the example graph. Starting from the main function, the total time in percent including all called functions is provided and how much of this time (in seconds) is spent in the function itself and in its called functions (children), respectively. In the following lines this timing information is provided for each child. For each parent of a function it is stated how much time a function contributes to it. Furthermore, gprof allows a line-by-line profiling for obtaining detailed results on a C source code line accuracy.

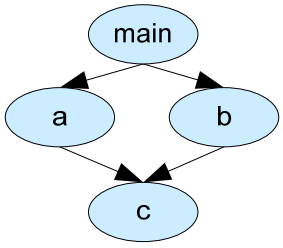



*Figure 4: Callgraph*

Gprof is a very useful tool for initial profiling. The drawback of the source code instrumentation is an inaccuracy, especially for small functions. The sampling method also leads to inaccuracy especially for short functions and only provides estimated values.

#### 2.3.2.2 Armprof

Armprof [14] is the profiler provided by with the ARM development environment. It provides results similar to gprof, namely a flat profile as well as a callgraph profile. However the results are restricted to percentage values of the overall execution time; see Listing 7 for an example result file.

```
Name                          cum%      self%    desc%     calls
main                         99.98%     0.00%   99.98%         1
  _printf                                0.00%    0.00%         2
  b                                     66.10%    8.66%        20
  a                                     16.54%    8.66%        20
-------------------------------------------------------------------
b                            74.77%    66.10%    8.66%        20
  c                                      8.66%    0.00%        20
-------------------------------------------------------------------
a                            25.20%    16.54%    8.66%        20
  c                                      8.66%    0.00%        20
-------------------------------------------------------------------
c                            17.33%    17.33%    0.00%        40
-------------------------------------------------------------------
```

*Listing 7: Armprof callgraph profile*

The operating mode of armprof differs from gprof. Profiling is not based on executing the program on real hardware, but running it on a simulator. The profiler is a built-in feature of the debugger armsd and applies the ARMulator, which is described in more detail in Section 2.2.1. Therefore, the code does not need to be instrumented. Only symbols need to present in the executable for identifying the function names.

Additionally to the basic cycle profiling, event-based profiling is provided by the ARMulator, i.e. the number of a specific event is counted for each functions. A wide range of events is supported including cache misses, interrupts and branch prediction failures. The results can be visualized with armprof in the same manner as the cycle profiling.

Similar to gprof armprof also applies a sampling-based mechanism and therefore encounters the same inaccuracy.

#### 2.3.2.3 ARM RealView Profiler

In 2007 ARM launched the RealView Profiler [8], a tool for profiling of ARM-based systems. It provides a graphical user interface (see Figure 5) and very detailed profiling results and is targeted to the optimization of software.

For this purpose it provides the following information besides the clock cycles:

- code coverage, including branch and statement coverage
- average cycle per instructions (CPI) and interlock information
- memory read and write accesses
- detailed graphical callgraph view and number of calls, callers and callees
- detailed information on function, class and source code line level including hot spots

For acquiring the runtime information for the profiling the RealView Profiler provides two options, either a high-level simulation or a hardware tracing. The simulation is implemented by means of the so-called Real Time System Model (RTSM), which are instruction-accurate

simulation models. They don't provide memory delay information, but allow an estimation of the execution time. A few RTSMs for ARM9- and ARM11-based systems are enclosed with the profiler, and further models can be built with a tool called System Generator.

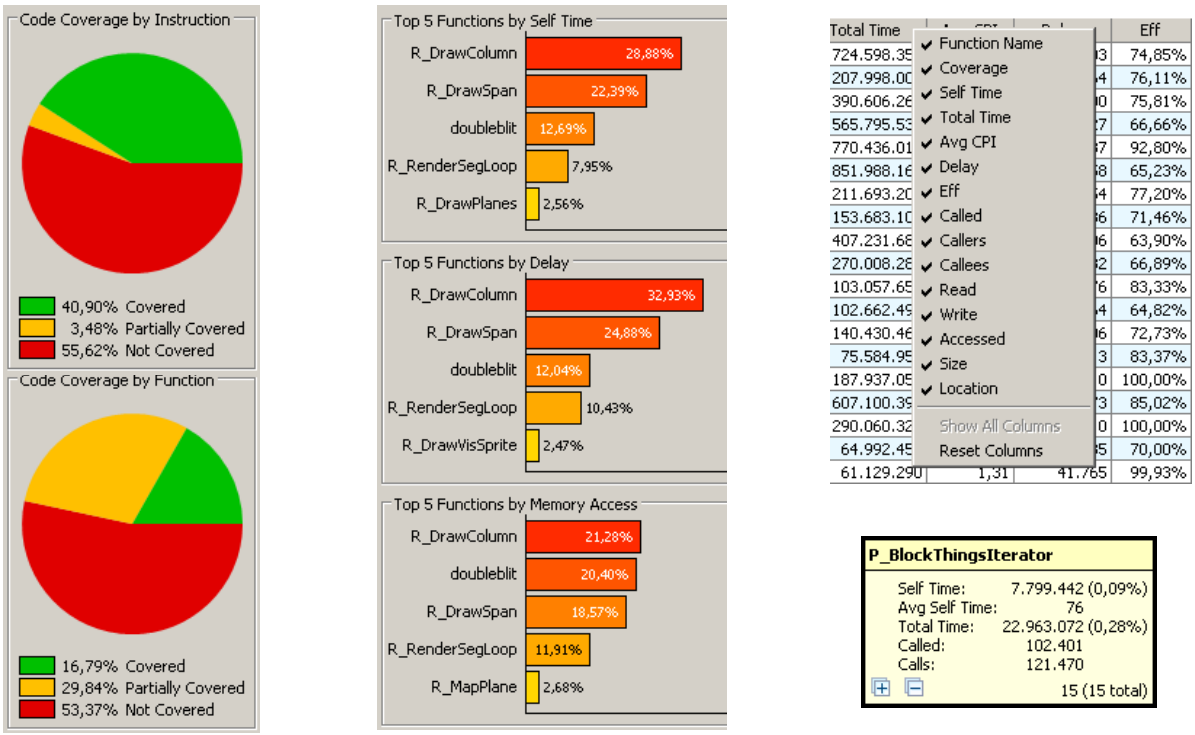



*Figure 5: Result GUI elements of the RealView Profiler*

The hardware tracing method is a sampling-based method, which employs on-chip tracing methods of the ARM processors, which is captured by an external device, namely RealView Trace. The tracing information is then parsed and analyzed by the profiler. Currently, the hardware tracing is restricted to the following three processors, ARM926EJ-S, ARM1136JF-S and ARM1176JZF-S.

The profiling results are very comprehensive; however when using RTSMs they lack timing accuracy of the processor and memory delay is neglected as well. Accurate timing can be accomplished with hardware tracing. This requires the appropriate hardware to be available, which hinders a broader design space exploration. Additionally, the sampling method lowers the accuracy. Memory profiling is only available in the sense of load/store operation tracing; the target memory (cache, internal or external memory) is not visible to the profiler.

#### 2.3.2.4 VTune Performance Analyzer

The VTune Performance Analzyer [57], similar to the RealView Profiler, is a GUI-based profiling tool. The tool is provided by Intel and supports their processor families, which includes for embedded systems only the XScale (PXA2xx) architecture respectively its successors from Marvell Technologies. VTune is a sampling-based profiler analyzing the program running on real hardware, usually the same machine as the profiler. The sampling is interrupt-based and can be triggered either by a specific event or by the timer. For PC processors a large set of events is supported and callgraph profiling is possible as well. For embedded processors the profiling is restricted to time-based sampling and a limited set of events, namely data buffer and dependency and branch-specific events. Embedded systems are profiled remotely by incorporating a data collector into the embedded operating system and visualizing the data

on the GUI running on a PC. The PXA processors have two to four hardware counters, the so-called performance monitor units (PMU), and the clock cycle counter for collecting the profiling information.

The interrupt-based sampling of VTune leads to a slight inaccuracy and the sampling of an application is executed while running on an operating system (OS), either Linux or Windows. This has the advantage of also reflecting the influence of the OS, however this is not always intended. Especially memory access optimizations are complicated, because of modified cache statistics due to task switches.

#### 2.3.2.5 Other Profilers

OProfile [64] is profiler for the Linux OS, which supports numerous processor architectures, including the XScale, MIPS and PowerPC embedded processors. Similar to VTune, it uses the hardware counters of the processor for data collection. Valgrind [71], also a linux-based profiler, provides very detailed profiling information, but only supports PC processors and the PowerPC embedded processor. Furthermore, most of processor development suites, such as ARC [6], CoWare Processor Designer [29] and Tensilica [41] deliver their own profilers, which are more or less comprehensive. For example the CoWare Processor Designer profiler delivers very detailed results including memory access statistics. However, all these profilers are restricted to the specific processor architectures.

### 2.3.3 The ATOMIUM Memory Profiler

The Belgian research institute IMEC developed a tool suite for optimizing data-intensive applications called ATOMIUM [25]. The tool suite consists of five tools, which focus on different aspects of memory-centric optimizations, including data reuse within a memory hierarchy, reducing the amount of data storage, memory architecture modifications and code pruning. The optimizations can be applied automatically by the tools and are based on an initial evaluation of the application. This first step within the optimization flow is carried out by the ATOMIUM/Analysis tool, which provides detailed profiling information of an application on C function level. For each function and for each variable within the C source code, read and write memory access statistics are generated. Additionally, a hot spot analysis identifying the most memory demanding code areas is carried out. The results are made available in different formats, for example as HTML pages. The profiling is based on instrumentation of the C source code, see Figure 6.

ATOMIUM/Analysis is a pure memory profiler, any timing information is omitted. It applies only a flat memory architecture during analysis. Both points restrict the usage of the tool for hardware tailored optimizations. The ATOMIUM/Memory Architect tool extends the memory architecture with parameterizable memory, which also supports timing information. Due to the instrumentation, the tool is dependent on source code availability, which prohibits profiling of 3-party libraries or applications. ATOMIUM is based on an abstract architecture model, thus the results are only an estimation of the real hardware results. This makes the tool well-suited for optimizations in an early design stage, where hardware independent optimizations take place.

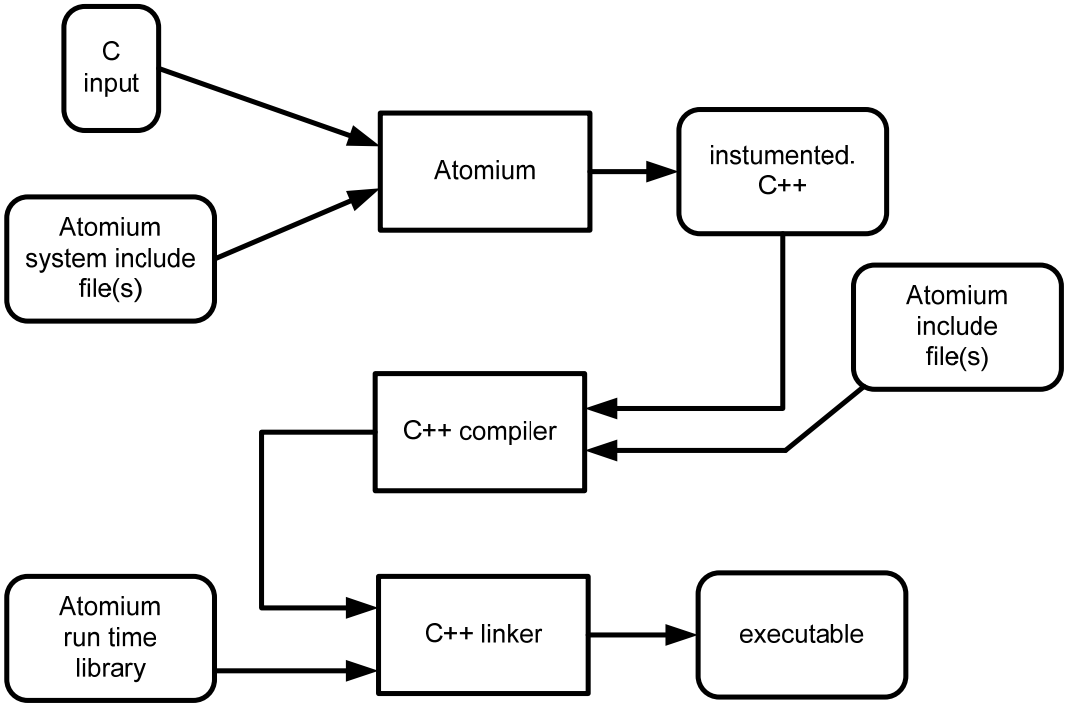



*Figure 6: ATOMIUM/Analysis toolflow [25]*

Besides ATOMIUM the most well-know memory profiling tool is Callgrind (an extension to Cachegrind), which is part of the Valgrind [71] tool suite. It provides very detailed analysis results, but is not available for embedded processors. At the Technical University of Munich a tool called iprof [62] was developed, which extends gprof profiling with instruction usage information. The tool performs a static analysis of the instruction usage of the program code on a basic block level, and correlates this information with the gprof sampling results. Thus an instruction profiling per basic block is accomplished. Memory access statistics can be generated by observing the usage of load and store instructions. Iprof is available only for the Intel-IA32 architecture and for SPARC processors.

### 2.3.4 Power Estimation Tools

Several tools exist for the estimation of power consumption of processors and embedded systems. Generally, they can be separated in measurement-based models and analytical models. JouleTrack [84] is a measurement-based tool, which provides several models on different levels of abstraction for two different processors. Numerous analytical models have been built on top of the processor simulator suite SimpleScalar [27]. PowerEscape [40] is a commercial tool which uses a parameterizable generic processor and memory architecture for estimation of power consumption, performance and memory accesses for a given application.

#### 2.3.4.1 JouleTrack

In 2001 a power estimation tool called JouleTrack [84] was developed at the Massachusetts Institute of Technology. It is implemented as a web-based application, which delivers energy consumption values for C source code that is uploaded to the web page. Two processor architectures are supported, the StrongARM SA-1100 and the Hitachi SH-4 RISC processor, two widely used embedded processors during that time.

Two distinct energy models are presented. Measurement of the current drawn while executing different instructions show a variation of up to 38 %, Figure 7 shows the result for the StrongARM processor. Contrarily, the average current for different benchmark programs varies only within a small range. The average current mainly depends on the supply voltage and core clock frequency. A simple first order model for energy estimation can be established as:

$$E_{tot} = V_{dd} \cdot I_0(V_{dd}, f) \cdot \Delta t \tag{1}$$

In this model the required energy only depends on the time required to execute the program, not on the actual program. The model estimates the energy within 8 % accuracy compared to the measurements of the tested programs.

As shown in Figure 7, the largest current deviation exists for the data transfer instructions load (LDxxx) and store (STxxx) and for the test (TST) and multiply-accumulate (MLA) instructions. Thus, especially for memory-intensive software, the first order model accuracy decreases.

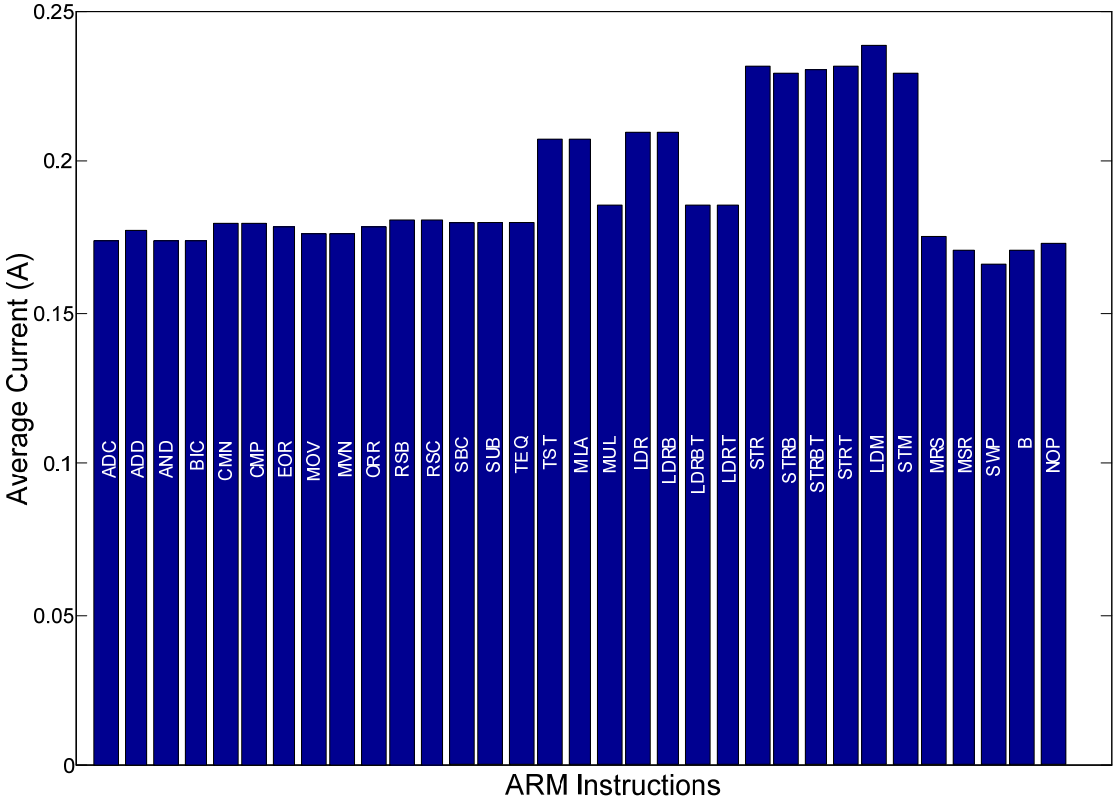



*Figure 7: Current values for instruction execution on a StrongARM SA-1100 [84]*

A second-order model incorporates the differences between the instructions and the states of the processor, especially for memory accesses. The instructions are grouped in several classes with similar power consumption and an average power value is then assigned to each group. Their influence is specified by weighting factors and the resulting current model is described as:

$$I(V_{dd}, f) = I_0(V_{dd}, f) \cdot \sum_{k=0}^{K-1} w_k \cdot c_k \tag{2}$$

The consumed current is defined by a base current $I_0$, which is the average current of all measurements and dependent on the supply voltage and the frequency. It is weighted depending on the weighting factors $w_k$ for the different instruction classes and on $c_k$, the portion of overall cycles consumed within this instruction class. The $c_k$ value for each class is determined based on instruction traces, which are created by running the program on a simulator.
For the StrongARM processor four instruction classes exist, instructions, sequential and non-sequential memory accesses and internal cycles, where the internal cycles and the sequential cycles lead to lower and the other two classes to higher current then $I_0$. The second-order model shows an accuracy of 2 %. The model has been further extended by the separation of dynamic and static power consumption, where the static part is due to the leakage current of the CMOS components.

JouleTrack give a good an accurate estimation for the energy consumption of the two processors. The results are restricted to the two processors presented and are only given for the entire program, not for every function. The tool is discontinued and no longer publicly available.

#### 2.3.4.2 PowerEscape

PowerEscape [40] is a power and performance profiling tool based on the ATOMIUM tool suite. It applies the techniques of ATOMIUM/Analysis and ATOMIUM/MemoryArchitect and is extended with power estimation capabilities. The simulated CPU architecture is parameterizable in terms of register count, bus width, CPU speed and further parameters and thus can be adapted to existing processors. The memory architecture can be customized including cache size, timing and replacement policy. Similar to the ATOMIUM tools the memory accesses are shown for each function and cache miss in hit reports. Additionally, timing and power information for memory accesses is provided. The tool can be used in batch mode, for example for finding an optimal cache size, and the results for accesses, time and required energy are presented as a Pareto curve.

PowerEscape was presented in 2004 by a newly founded company with the same name. The current status of the tool is unclear, as the company was acquired by ARM Ltd. in 2006 [16]. So far, the PowerEscape is not available as a single product from ARM.

#### 2.3.4.3 Analytical Models for Power Estimation

Several cycle-accurate analytical power models were built on top of the SimpleScalar [27] processor simulator suite. SimplePower [102] creates an analytical energy model of the processor. A register transfer (RT) level model of the processor is used for this purpose, which is built of basic hardware blocks, such as multiplexers, latches, adders and shifters. These blocks are accompanied by power values depending on the Hamming distance of their input pattern. The RT level model is triggered by the processor simulator and thus the switching activity of the basic blocks is simulated and the energy consumption can be estimated. As an analytical model it can be adapted to different technology feature size by exchanging the basic block power value library. The RT-level model of the processor core covers the entire pipeline and the register file, but neglects the control logic and clock distribution tree. For cache simulation SimplePower applies the DineroIII [31] cache simulator enhanced with analytical energy models. Other internal or external memory is not modeled.

A similar approach was taken by the developers of SimpleScalar themselves. The SimPanalyzer extends the SimpleScalar simulator with a highly elaborated power model simulating the processor architecture on gate-level with a cycle-accurate timing. Similar as in Sim-

plePower, the processor is divided into basic blocks. However, a special emphasis is put on the clock distribution tree, which can have a huge influence on the overall performance. Beside the input switching activity the internal structure of these blocks is considered, including gate and interconnect capacitance and resistance. These values are calculated based on technology parameters and user defined parameters about the architecture of the basic blocks. The clock tree, for example, is specified by die area, the clock skew and the clock node capacitance. For cache power simulation either CACTI [96] models can be applied or the more accurate models provided by the tool, which are based on detailed SRAM and address decoding models. These SRAM models are also used for other memory components of the processor, such as a register file or a table look-aside buffer. The models are calibrated against SPICE (HSPICE) simulations of the components and show an average estimation error of 7 %. The tool provides models of the StrongARM and ALPHA processors.

The Wattch [26] extension to the SimpleScalar simulator also implements a similar power model. Wattch also uses CACTI and performs a detailed clock tree modeling. However, other parts of the model are not as detailed, for example the power consumption model of the combinational logic in the data path is based on scaled power values of similar structures in different technology. Models are provided for an Alpha, Pentium Pro and a MIPS processor.

#### 2.3.4.4 Functional-Level Power Analysis

Qu et al. [77] present a higher (function)-level approach for power estimation of processors. Based on the assumption that many programs spend most of their execution time in a few functions, power modeling of these functions allows a decent estimation of overall power consumption. As an example, they name floating point library functions. The power models for these functions can be stored in a data bank. The power consumption of a user application is estimated by tracing its execution on a simulator and counting the number of library function calls. The number of calls of each function can then be multiplied with the associated power value in the data bank and then each of these sums is added together to achieve the overall power consumption. This approach is restricted in numerous ways. First of all, the approach is unfeasible for applications that are not implemented based on library functions. For example, the implementation of a video decoder, as presented in Chapter 6.1, uses only optimized hand-written C code. Additionally, only static power values are stored for each function in the data base. This can not reflect real-time behavior influences, e.g. cache misses or conditional code execution.

Julien et al. [61] apply functional-level power analysis (FLPA) to model power estimation of a complex DSP processor. Contrary to the instruction-level power analysis (ILPA) proposed by Tiwari, FLPA models the power consumption on a higher level of abstraction. The model is based on the condition that processors can be divided into functional units and the power consumption of each unit can be described by a few parameters. Contrary to the analytical models presented in the previous section the units are on a higher level of abstraction. Instead of using units on an RT level, such as registers or multiplexers, functional blocks are used, such as a processing unit or DMA controller. This has the major advantage that a detailed knowledge of the internal processor architecture is not required. The activity of each unit can be described by a few parameters. In the approach of Julien et al., two sets of parameters are defined. One set of parameters (algorithmic parameters) is defined for describing the dependency of the model on the software activity– e.g. cache misses and degree of parallelism. The second set includes system-specific dependencies (architectural parameters), which include the memory setup and the core clock frequency. Instead of incorporating all profiling details

of the instruction trace of an application, only a few parameters need to extracted and fed to the model.

The coarse-grain nature of the approach makes it very suitable for complex processor architectures, such as VLIW processors. Applying ILPA models would require a huge number of measurements, as every combination of instructions as VLIWs would need to be modeled, as well as any sequence of VLIWs, to cover all inter-instruction dependencies. The advantage of the FLPA methodology is the lower complexity of the modeling and estimation, but this comes at the cost of lower accuracy.

Blume et al. [23] note that the FLPA approach delivers an appropriate accuracy for only a specific set of applications with small power dynamics. Therefore they extended it with an instruction-level model to form a hybrid FLPA/ILPA model. The methodology has been proven to work on an ARM940T processor as well as on a heterogeneous SoC architecture incorporating an ARM926EJ-S and a Texas Instruments C55x DSP core. The FLPA model for the ARM940T is comprised of three functional blocks: the processor core, the data and the instruction cache. For the ARM926EJ-S, additionally the on-chip SRAM and the external SDRAM are incorporated into the model. During the FLPA model creation it was found that the power consumption is fairly dependent on the executed instruction, which leads to the necessity of incorporating instruction-level analysis to increase the accuracy. For power estimation based on this model, this in turn means that a profiling of the user application is required to reveal the dynamic instruction distribution. For the ARM940T, the instructions can be clustered in six groups with similar power consumption, which reduces the complexity of the model. Similar models were built for the ARM926EJ-S and the C55x DSP. With these models Blume et al. were able to provide a high estimation accuracy of the models for a wide range of applications. Especially for the heterogeneous SoC they achieved high accuracy with a maximum error of 3.6 %.

# 3 Theoretical Background

This chapter gives an overview of the underlying theory and techniques of profiling for embedded systems. At first, the different components of such systems are examined, including the processor, memory and bus architectures. Subsequently simulation models for these components are surveyed, which are typically used for testing their functionality, and more important for profiling, their timing behavior. Different techniques exist for profiling the execution of applications on a processor. They differ for example in their level of accuracy, speed and level of details. Some established software optimizations and hardware architecture decisions are presented, which typically benefit from the knowledge gained during profiling, for example by finding the right positions in the source code for applying the code modifications. Finally an example of a demanding data-intensive application, the H.264/AVC video coding standard is presented.

## 3.1 Embedded System Components

Embedded Systems are usually composed of multiple functional units, which overtake different tasks within the data processing system, such as data retrieval, transport, processing and storage as well as control purposes. Usually the central processing and control unit of an embedded system is a microprocessor or a microcontroller. Program code and data are stored in a hierarchy of memory units, which cover the different storage needs concerning live time, access time and size. Data transfer between processing, storage and I/O units is often provided by bus systems, as they allow an efficient manner of interconnectedness. If high speed data transfers are required, the bandwidth of busses might not be sufficient and dedicated transfer channels are used instead, such as FIFO buffers. If the processing power of a single processor is not sufficient, the system is extended by further processing units. Either the same processor is instantiated multiple times leading to a homogeneous multi-processor system, or specialized processing units, such as DSPs or coprocessors based on dedicated logic are added to the system.

### 3.1.1 Processor Architectures

In order to work within the constraints of embedded systems, processors with a small and simple architecture are employed. In the 80s the Reduced Instruction Set Computer (RISC) architecture was developed, which allows the design of fast and small processors and has become the primary choice for microprocessors in embedded systems. Depending on the application field and the required performance processors with a data path width between 4 and 128 bits are used and clock rates from a few MHz up to 1 GHz are available. In the field of data-intensive applications, the focus of this work, usually processors are applied, which are at least 16-bit wide and run with more than 100 MHz. Section 2.1.2 gives an overview of the most common processor architectures for this application area.

RISC processors feature a simple instruction set as compared to the Complex Instruction Set Computer (CISC) architectures, which is preferred in personal computers. The major goal of the RISC architecture is to execute each instruction within the same amount of time, e.g. one clock cycle, and have a simple, equal-length, instruction format. Memory accesses usually require much more time than data processing functions, therefore memory accesses are only allowed by special load and store instructions, which also leads to the name load/store architecture. Other instructions can only be applied on data, which resides in the local register file and can thus be executed very fast.

This simplicity is the basis for applying pipelining mechanisms, which allows a speedup by parallel processing of several instructions at the same time. With pipelining the processing of each instruction is divided into several steps, so-called pipeline stages, with a similar length required for their execution. Figure 8 shows a pipeline with five stages, in the first stage an instruction is fetched from memory, or if available, from the instruction cache and stored in the instruction register (IR). The instruction is fetched from the address given in the program counter (PC). The PC is then either automatically incremented to the next instruction address or updated with a new value, e.g. if a previous branch instruction requests it. The instruction decoder evaluates the instruction in the next state and generates the signals required for controlling the other processor components, e.g. within this stage source register values are transferred from the register file to the ALU source registers (S1 and S2). Then the actual instruction execution takes place, for example a SUB operation the ALU subtracts S1 from S2 and stores the result in the result register (RES). For load and store operations the address calculation is performed in this stage, and the actual data access to the memory (or data cache) is performed in the next stage. In the final stage, the ALU result or the data from memory is written back to the register file.

The pipelining allows the parallel execution of multiple instructions by starting the processing of a new instruction while other instructions are still in the pipeline. Usually the pipeline advances by one stage each clock cycle, which leads to an instruction per cycle (IPC) count of one, i.e. each clock cycle one instruction leaves the pipeline. The IPC decreases if a pipeline stall is required, which occurs if data dependencies between instructions in the pipeline are present or if the execution on an instruction or a memory access requires more than one cycle. The IPC also decreases by branch instruction, which requires a flush of the pipeline.

The pipeline length of current embedded processors ranges between three and nine stages. The longer pipelines allow a higher clock frequency and thus lead to an increased instruction throughput. At the same time the negative influence of pipeline flushes on the performance also increases. Branch prediction mechanisms are introduced to such architectures in order to reduce the need of flushes by filling the pipeline corresponding to the expected program flow. In the following sections some examples for different pipelines are given.

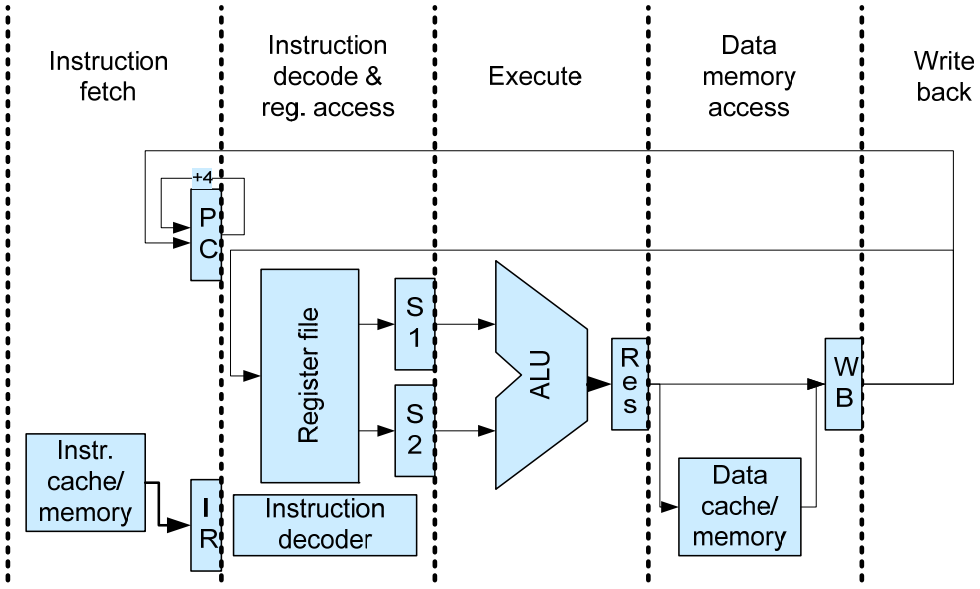



*Figure 8: Typical data path of a RISC processor with a 5-stage pipeline*

The fact that data processing is restricted to data available in the register file is a huge burden especially for data-intensive applications. For every load or store to memory an extra instruction needs to be issued. In order to reduce the number of these instructions RISC processor usually come with a large set of registers, thus values can be kept in registers as long as required for processing. If more data are required than can be stored in the register, a so-called register spill is required, in order to make place for new data. Advanced memory architectures, as described in Section 3.1.2, are applied to reduce the time required for register spill and fill. The drawback of a huge amount of registers is the large die area required by the register file and the addressing size required in the instruction format, e.g. with 64 registers and a 3-address format 18 bits of the instruction word are taken already for register addressing. Furthermore, when a function call occurs, a larger number of registers need to be saved on the stack. To overcome this problem register windowing has been introduced, as used for example in the Tensilica [41] or SPARC [38] architectures.

In many data processing algorithms the width of the data is lower than data path width available in the processor, for example when processing 8-bit video data on a 32-bit processor. Furthermore quite often the same operation needs to be performed many times to adjacent pixels. This observation leads to the Single Instruction Multiple Data (SIMD) architecture [34], where the data path is separated in multiple data paths, e.g. the 32 bits are split into four times 8 bits, and on each of these paths the same operation is applied. In best case this leads to a four times increase performance, however data may need to be rearranged by special instructions in order to fit into SIMD format, which reduces the speed- up factor. SIMD is especially used in multimedia instruction set extensions for embedded processors, e.g. the Wireless MMX in XScale, or in Digital Signal Processors (DSPs). As SIMD is basically an organizational feature it has a low impact on the hardware requirements. It mainly requires some additions in the control unit for separating the data path and for extending the instruction decoding.

### 3.1.2 Memory Architectures

The instructions and data to be processed by the processor are stored in memory. The most important characteristics of memory devices are their size and speed, which are influenced by the applied technologies and architectures. Processor-based systems are typically connected to a hierarchically structured memory architecture, see Figure 9, starting with a fast and small memory and ending with large and slow storage devices. Memory has an enormous influence on the overall system architecture, in terms of power consumption, die area and performance.

Permanent storage of program code and data is usually provided by disk memory, such as hard disk or DVDs. In embedded systems flash memory is often used for this purpose. Disk memory is slow; the access time is in the region of milliseconds and the transfer rates below 100 MB/s. Therefore intermediate storage is provided in volatile memory devices, such as dynamic RAMs (DRAM), with access times in the range of nanoseconds and transfer rates up to a few gigabytes per second.

Until the early 1980’s DRAM devices and processors achieved about the same speed. Since then the processor speed has increased every year by a factor between 1.35 and 1.55, whereas DRAM only showed about 7 % improvement in latency per year [45]. This expanding gap leads to the necessity of intermediate storage between memory and processor, in order to bridge the speed difference between them. Over the years, a hierarchy of memory units was developed to close the gap, namely multiple levels of caches and tightly coupled memory (TCM). The fastest storage element in the memory hierarchy is the register file.

The underlying principle of the memory hierarchy is to have those parts of the data and program code available in faster intermediate memory, which are expected to be used during the following program steps. The basic idea is to take advantage of the spatial and temporal locality within data processing and program execution. The fact that most of the program code is executed in sequential manner and stored in the same way in memory, leads to spatial locality. Temporal locality occurs in loop execution, where the same sequence of code is executed many times. From the data processing perspective spatial locality occurs, when neighboring data in memory is accessed consecutively, e.g. when filtering each pixel of a frame. Temporal locality can also be found frequently in data processing; variables are often used multiple times during a calculation. Besides locality, prediction mechanism can be used for optimizing memory accesses, as for example used in branch prediction and data pre-fetching. The principle of locality and prediction can either be used during compile-time or run-time.

#### 3.1.2.1 Register File

The register file is directly connected to the ALU and provided the fastest access to data. It contains numerous registers, and is either implemented as fast SRAM cell or assembled of flip-flips and allows access within a single CPU clock cycle. Registers can be directly accessed by numbers or are arranged in sets (windows), with one set of registers visible at a time. Registers are assembled of flip-flops and are expensive in terms of die area and power consumption and also the allocation process. The allocation of registers with data values is determined by the compiler, and thus is static during run-time. The register allocation is performed with regards to the temporal locality of the data values. The counterpart to the register file for fast program code access is the instruction register.

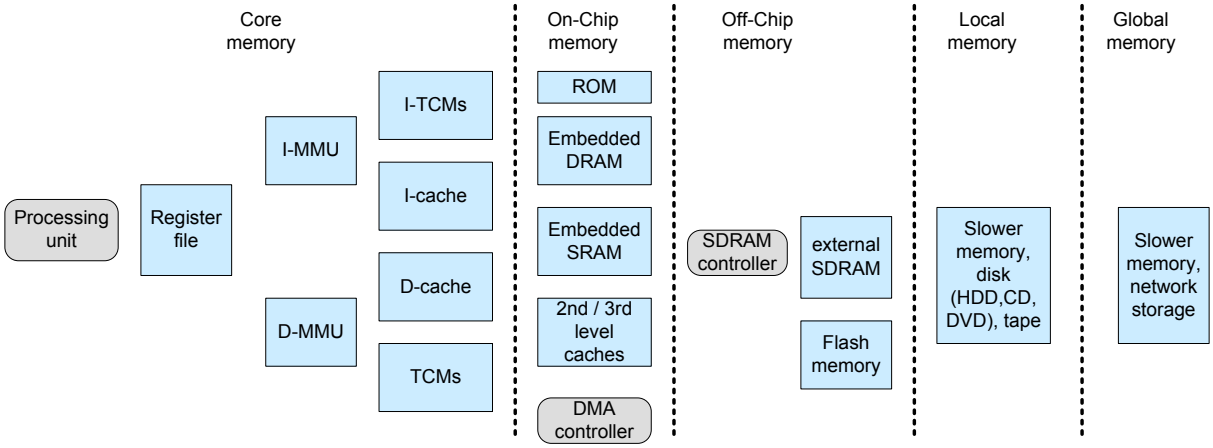



*Figure 9: Typical memory hierarchy in computing devices*

#### 3.1.2.2 On-Chip Memory

A **cache** is a fast memory, which stores recently used data and program code. When a data value is requested by the CPU at first a cache lookup is performed. If data can not be found in the cache it is loaded from memory to the CPU and also stored in the cache, usually including some data at neighboring addresses. Access times for caches are within a range of one to a few CPU clock cycles.

Caches are organized in sets and lines, see Figure 10. Each set consist of a tag RAM and a data RAM, which in turn contains a number of lines, e.g. 256 lines. The tag RAM contains a part of the memory address (the tag) of the data values in the corresponding data RAM line. A

line in the data RAM is used to store a data value and its neighbors and has a size of a few words, for example eight words. If the CPU requests a data value from the cache, the data address is spilt into a tag, index, word and byte part. The index part corresponds to a specific line, the word part identifies the word within a line and the byte part selects the byte. The rest of the data address is the tag. The tag is compared with the value stored in the tag RAM, and if they are equal the requested data can be read from the data RAM. Otherwise the requested data value needs to be read from memory and is together with its seven neighbors in the line. Caches with only one set of tag and data RAM are called direct-mapped caches. Such cache architectures are simple, however they show a significant drawback. Due to the index and word parts of the address, each data address can only be mapped to one specific location in the cache. If this location is already in use when loading data from memory into the cache, the entire line containing this location is overwritten with the new line. This decreases the performance significantly, when two data values from different memory addresses, which are mapped to the same line, are required alternating. In order to overcome this drawback caches with multiple sets, so-called set-associative caches, have been introduced. In the example four sets, and thus four location candidates are available for each data item. The decision, which set is used to place new data, is taken upon more or less advanced mechanisms; a very common policy replaces the least recently used (LRU) line. Set-Associative caches require more die area, because for each set a comparator and additional control logic and multiplexers are required.

Caches are very efficient hardware structures for taking advantage of locality in data and program code during runtime. Loading an entire line to memory, instead of only the requested memory address, severs the principle of spatial locality, whereas the associativity increases the temporal locality advantage.

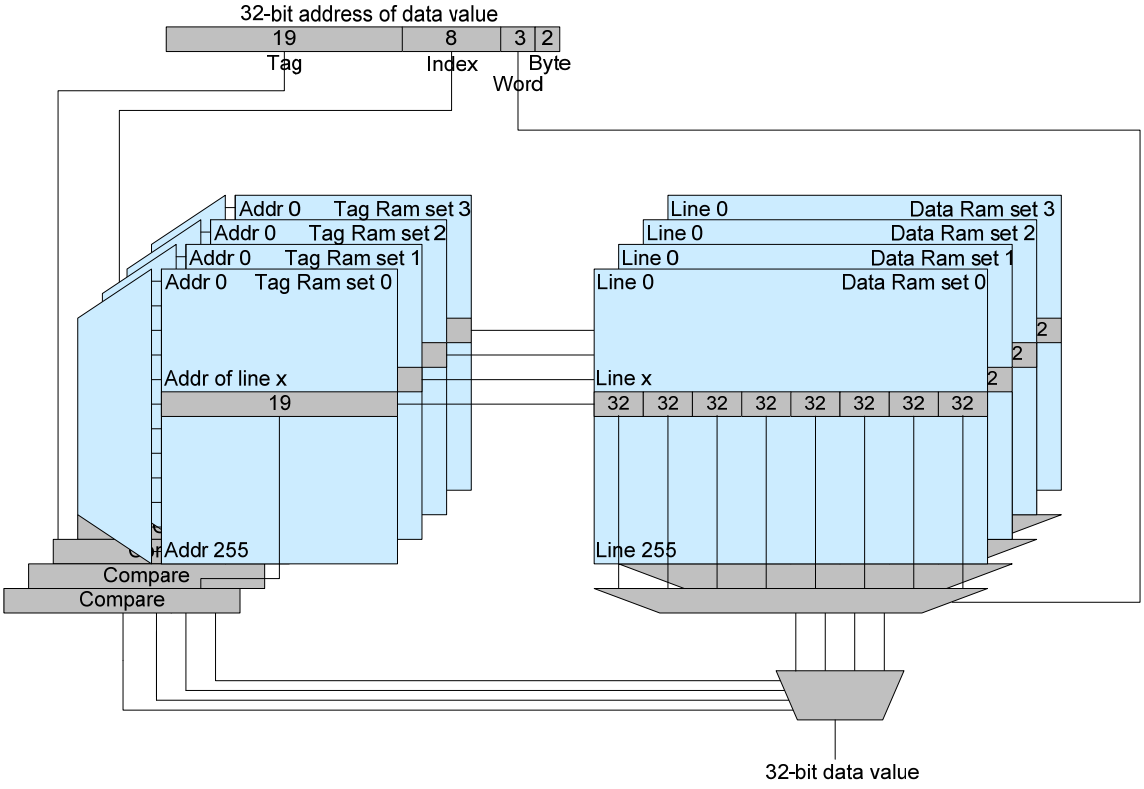



*Figure 10: Architecture of a four-way set associative cache (eight words per lines, 255 lines per set)*

Caches keep copies of data, which actually resides in memory. If the memory content is changed, cache coherency problems occur. This happens if other system components, e.g. other processors, DMA controller or hardware components have write access to the memory. In such cases the cache content needs to be refreshed (invalidated) in order to synchronize the memory and the cache content. Furthermore, especially in real-time critical embedded environments, the unpredictable timing behavior that caches introduce to the system can be problematic. The concept of caches has been extended to multiple levels of caches, which are connected in a hierarchical manner, from smaller and fast, usually multi-set associative, to larger and slower direct-mapped caches. Caches contribute highly to power consumption and die area. For example, depending on the technology, the two 8 kB caches of the ARM946E-S processor require between 40 % and 60 % of the die area and are responsible for 30 % of the power consumption [8].

**Write buffers** are intermediate storage devices between the CPU core and the bus/cache. They are used for avoiding the wait states required when storing data from the CPU to the main memory. Instead of writing directly to the memory the data values are stored together with their address in the write buffer, which only requires one CPU clock cycles. The write buffer then performs the actual store operation independently and the CPU continues without a pipeline stall. Write buffers usually have a size of a few words, e.g. 16 for the ARM9E family of processors, and data does not reside in the cache or if the write-through mode is used.

**Memory management units (MMU)** are used to control the memory access of the processor. Among other tasks, the MMU typically performs address translations, i.e. the MMU translates every (virtual) memory address from the CPU to physical addresses on the memory bus. Operating systems utilize address translation in order to provide a consistent memory space to all running processes, e.g. all processes use a virtual address space starting at address 0x0, whereas the physical addresses are mapped to distinct locations. Furthermore, the access behavior for each memory area can be specified, this includes access restrictions, such as read- or write only access and if caches are used for this area. If a forbidden memory access occurs the MMU triggers an interrupt, which simplifies the debugging process. Non-cacheable memory areas are used for memory locations, which can be modified by other components than the CPU, for example status registers or output memory of hardware components. Otherwise, cache inconsistency may occur if the cache holds an old copy of the memory data. If tightly coupled memory is available in the system, the mapping of this memory to the address space is also provided by the MMU. For the address translation the memory space is divided into pages, e.g. 4 kB per page, and page tables provide the physical base address of a page that corresponds to a specific virtual address. The page tables are stored in a fast memory called translation lookaside buffer (TLB) and can be modified by the operating system.

**Tightly coupled memory (TCM)** is on-chip memory, which allows fast data or program code access. It is located close to the processor and is usually built of SRAM cells, which run with the same speed as the processor. Its main purpose is intermediate storage of frequently accessed data and program code. A typical example would be a data array, which is accessed in a loop. During loop execution this array should be stored in the TCM. Instruction TCMs might be used to store interrupt routines, in order to allow a fast serving. The major advantage over caches is that the behavior of TCMs is predictable, as the content is controlled by the software. Data replacement in caches can lead to a cache trashing, i.e. two consecutively accessed memory addresses share the same line in cache, such lead to a cache miss, which incurs a write back and new data fetch. Considering the loop execution example, often multiple data arrays are accessed, as for example in the following code [74]:

```
int a[N], b[N], c[N];
```

```
…
for i in 0 to N-1
c[i] = a[i] + b[i];
end for;
```

*Listing 8: Simple loop showing the risk of cache trashing*

If the size of these arrays is a multiple of the set size of the cache, and the arrays are arranged consecutively in memory, than a[i], b[i] and c[i] map to the same cache line. Loading one of them replaces the former one in the cache; this behavior is called cache trashing and leads to an enormous performance decrease. Furthermore caches require more chip area and energy than TCMs due to the TAG-RAMs and comparators. On the other hand TCMs need to be controlled by the software, which may leads to overhead during filling or writing back from the TCMs. If a DMA controller is available in the system, it can overtake the burden of the data transfers. The size of TCMs is similar to cache sizes, usually between a few to some hundreds of kilobytes. Some processors implement fast memory by assigning a fraction of the cache to the purpose of TCMs [3]. If both exist, TCMs are placed parallel to the cache, and the MMU controls the access depending on the memory mapping.

In embedded systems the previously mentioned core memory components are extended with on-chip and off-chip memory and controllers. These memory units are larger than the core memory units, but require access times of multiple processor cycles. On-chip **SRAMs** are built of cells that a composed of six transistors. These transistors form a flip-flop with two stable states for storing one bit. The cells are connected via the word-lines to the address decoder and the two differential bit-lines allow data write or read. The SRAM is organized in rows and columns, all cells of the same row share a word-line, and all cells in one column are connected to the same bit line. Before a read operation occurs, the bit lines are precharged to a voltage level in the middle between the “0” and “1” level. Especially for large SRAMs the long bit lines with a high capacitive load lead to high power consumption due to the precharging. On-chip SRAMs are operated with processor speed or a small divider of the processor speed, depending on the internal bus architecture. SRAMs may also be replaced by **embedded DRAM** cells [58], which are built of regular one-transistor DRAM cell, but provide an SRAM-like interface. Embedded DRAMs offer a drastically decreased die size and lower power consumption, but provided a lower access speed. Furthermore, due to the different technologies, integrating embedded DRAM with logic circuits on one die requires higher design effort.

Especially application-specific systems provide on-chip **ROMs**. They are used for storing program code, e.g. the boot-up sequence of code, or constant data values, such as look-up tables for encoding or filtering.

#### 3.1.2.3 DMA Controller

For data-intensive applications, a large portion of the processing time is spent with data accesses. This can be either for moving data within memory or for waiting for data transfer from slower memory. In order to allow concurrent data transfer to data processing, a specialized data transfer unit, a direct memory access (DMA) controller can be used. **DMA controllers** are bus master components, which can transfer data from one memory location to another. The transfer is initiated by the CPU (or another component) but takes place autonomously from the initiator. In order to use the DMA controller efficiently it should be controllable in a “fire-and-forget” manner, i.e. the initiator should be able to continue work as usual after filling the transfer task to the DMA controller, and only be informed when the transfer is accomplished. This is either done with an interrupt signal or with status information in the DMA

controller. In order to allow this, DMA controllers have multiple slots, so-called channels. Each channel carries the information of one data transfer, which consists of the source and destination address and the amount of data to be transferred. The number of channels it either fixed, e.g. 16, or it can be extended by channel chaining, i.e. the channel information is stored in form of a linked list in memory. The channel initialization and the transfer initiation can be performed separately. This is especially helpful if a specific transfer need to take place regularly, for example for copying the video frame buffer from main memory to the video output device. Also some DMA controllers allow auto-increment of source or destination addresses after a transfer has finished. Especially for video processing two-dimensional DMA transfers are very helpful. For such transfers the channel information is extended with width, height, and a stride dimensions. Figure 11 shows a typical two-dimensional DMA transfer operation.

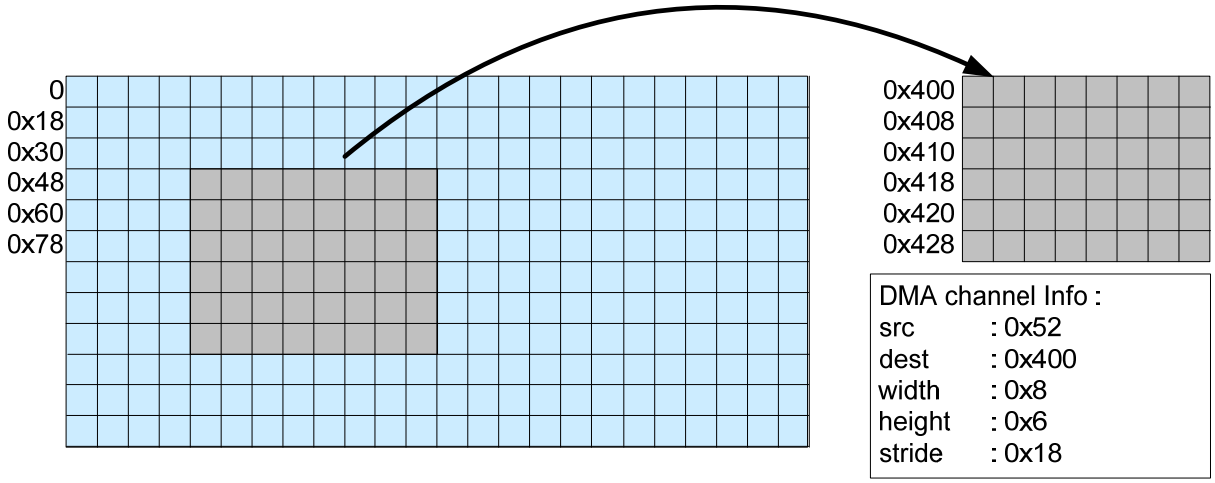



*Figure 11: Two-dimensional DMA transfer*

#### 3.1.2.4 Off-Chip Memory

External memory is used for storing program code and data, such as heap and stack. As non-permanent memory either SRAMs or DRAMs are used, whereas DRAMs are much cheaper. Typical DRAM components are synchronous DRAM (SDRAMs) [54], which have a clocked input/output behavior.

**SDRAMs** are organized in banks, for example two or four banks, where each bank can be controlled separately. Many SDRAM commands, such as active, pre-charge or read, need multiple cycles to be executed. The bank layout allows multiple commands to be executed concurrently, and effectively increases the performance. Each bank is organized in rows (also called pages) and columns, similar to SRAMs. The access of a specific memory cell requires multiple steps. At first a page has to be activated during the row-address strobe (RAS) phase. In the following column address strobe (CAS) phase the cell within the activated page is selected, and the read/write signal indicates the operation. For read accesses the memory requires two to three cycles (CAS latency) before the value is available on the bit lines. During this time the bank can accept new commands, for example the next read command. In order to decrease the required commands burst modes can be used, which automatically read or write a number of consecutive memory cells. Before the next page can be opened the current page needs to be closed, which implies a pre-charge of the bit-lines.

All of these features, namely the parallel bank access, the pipelining within each bank, and the burst modes lead to a higher throughput. They help to cover the slow access time of DRAM cells, which is about 60 ns, and allow data transfers on every clock cycles even for 6 ns cycle

times (167 MHz) and thus reach the performance of SRAM. The SDRAM performance has been enhanced with the double-data-rate (DDR) concept. DDR SDRAMs have a data I/O buffer extended with a small buffer for storing multiple data values. In the so-called prefetching multiple consecutive data words are read in one clock cycle from the memory bank to the buffer, and transferred in two half-cycles sequentially to the data output pins. In burst transfer mode this can increase the actual data rate up to factor equal to the number of concurrently read words.

The disadvantage of SDRAMs as compared to SRAMs is that they are far more complex to operate. SDRAM controllers are required which contain large state-machines to reflect the internal and pipelined behavior of the memory. Furthermore as the capacitors loose their charge over time, a periodical refresh every 50 - 100 ms is required. During this phase the memory is not accessible. In case of random access to the memory, many page misses occur, and the required activation and deactivation time decrease the performance.

Due to their complex structure, SDRAMs are also more difficult to model in a simulation environment. The ARMulator, see Section 2.2.1 uses a simplified model instead, which only differentiates between sequential and non-sequential accesses.

Contrary to desktop PCs, embedded systems often use **flash memory** for permanent (non-volatile) data storage instead of hard disks. This is due to the fact that embedded systems have lower demands on the storage space and that they are used in an environment, where hard drives might be too sensitive. Flash memory has a simple interface similar to the SRAM interface. The access time for read and write operations differs in two orders of magnitude, whereas read operations are in the range of DRAM access times, about 120 ns for single access, write accesses are much slower.

The power consumption of SDRAM and flash memory are very different. Flash memory has a much lower power consumption especially when put into standby mode. The Intel flash device used in this work [55] is specified with 120uA current during standby, whereas an SDRAM [79] memory requires about 30 times more (4 mA). For active memory accesses the difference is not as drastic, a read operation requires on the flash memory 30-50mA, a write access 80mA, the SDRAM dissipates 150 - 180 mA for both, read or write.

### 3.1.3 Interconnection Architectures

The interconnection of components in an embedded system has a crucial impact on the performance. The most important interconnect in processor-based systems is between the processor and the memory. Processors require instructions and data for operation, whereas fast instruction transfer is the most dominant performance factor, because whenever instruction transfer stops the processor must be halted. From the processor view one can differentiate between **von Neumann** and **Harvard architectures**, since the former combines data and instruction transfers on one bus, while the latter utilizes separate busses. If separate memory devices are available for instruction and data, such as instruction and data cache, the Harvard architecture increases the performance of data-intensive applications significantly. Besides the basic data interface (load/store interface) some processors offer further choices for data supply. Some processors [6] offer multiple load/store interfaces, which allow for example the X and Y memory interfaces for DSP operations. User-defined register file extensions [6] can also be used as a secondary data interface or specific FIFO ports [94] that allow direct connect to other system components. Usually the usage of these architectural features requires hand-optimized assembly programming, as they are not automatically utilized by compiler for code generation.

The processor, respectively the cache, is connected to the other system components, which includes memory, I/O components and further (co-) processors. The interconnection is characterized by two major aspects, bandwidth and latency. The simplest, and in embedded systems most used interconnection architecture, is the **bus**, which connects several components to a collection of shared physical wires (shared-medium network). The communication on the bus is initiated by one component, the bus master, and is directed to one or more other components, the bus slaves. At any time only one transaction can take place on the bus. This makes it very suitable if only one master and an arbitrary number of slaves exist on the bus. Multi-Master busses require an arbitration unit, which assigns the bus with a given prioritization scheme to the masters. If many masters compete for the bus, long waiting periods can occur and reduce the overall performance. Also if slaves have long latencies, if a read page miss on an SDRAM occurs, the bus is occupied until the slave can serve the required access. Thus the available bus bandwidth can not be used efficiently. Different mechanism try to overcome this problem, for example split transactions, pipelining, multi-channel busses, see Section 2.1.4 for detailed description. Busses have the advantage that they can be implemented and extended easily and are very area efficient due to the shared medium concept. In large chip designs, with many components and long wires, single bus architectures are energy inefficient and slow due to their broadcast behavior and the high capacitance and cross-talk between the wires. Advanced interconnect architectures have been developed for these so-called network-on-chips (NoCs) [63].

If multiple masters and many slaves are present in the system, such as in **network-on-chip (NoC)** [63] architectures, more advanced interconnections are useful, in order to allow parallel transfers. Crossbar, mesh or ring interconnection [43, 75] are used for such purposes. These networks reflect the architectural features of computer communication networks, such as dividing the network in sub-networks for fast local transfers, connecting these sub-networks with switches and routers and sending data in packages. An example of such large-scale switch-based networks is an FPGA. This programmable device features a large array of processing elements, which are connected by a hierarchical network of busses.

Due to their reduced feature set and application field, SoC in embedded systems usually only use simple interconnection schemes, such as single- or multi-bus architectures. Multiple busses are used to either separate the slow from the fast components or to enlarge the available bandwidth in multi-master arrangements. Slow components, such as serial I/O interfaces, can be arranged on simple busses, i.e. with a smaller bit-width and single master concept, in order to reduce the area requirements. Typical examples for on-chip busses are AMBA [10], wishbone [73], CoreConnect [51] and STBus [80]. Some of the busses have different specifications for slow and high-speed busses, such as the widely used AMBA busses, which are described in Section 2.1.4 in more detail. Off-chip interconnection in embedded systems is often not standardized, and instead designed for the needs of the system, contrary to PC motherboards, where several bus standards are defined, such as HyperTransport, QuickPath or PCI-Express. External components, such as SDRAM, flash memory, network or video controller chips are either connected to special purpose I/O pins of the chip or connected to a bus. In order to reduce the complexity, the bus protocol is often simple and asynchronous, using hand-shaking for synchronization. Furthermore the required wires are reduced by using a single tristate bus for read and write data and by multiplexing address and data bus on the same wires. Depending on the size of the PCB, the off-chip wires can have a high capacitance, which needs to be driven by the I/O drivers and thus influence the power consumption.

## 3.2 Simulation Models

The profiling of an embedded system can either be performed on the real hardware or on a model of the system. The advantage of the real hardware is that the results are highly accurate. However it also requires that hardware is already available, and does not allow variations of the system. Therefore if the profiling should be in an earlier design phase, for example for design space exploration, models for all parts of the embedded system are required. Depending on the type of profiling that is requested the models may vary in their reflected accuracy and features. The processor models for simulating the software execution is the most vital element during the profiling and is sufficient for simple systems. For more complex and memory-centric systems a memory model should also be used. Besides the modeling of the timing behavior, the energy consumption can also be considered during the simulation. Additionally to the processor and the memory subsystem other hardware components, such as co-processors or I/O components might influence the system behavior significantly. In this case a cosimulation system [50, 44] is required in order to allow a realistic profiling scenario.

Beside the timing and memory access behavior an estimation of the energy or power requirements is helpful within the embedded system design flow. Whereas the power consumption is more interesting for the thermal design and the peak current drawn from the battery, the consumed energy is relevant for software optimization and battery run time, as it also considers the execution time. The energy estimation can be done by means of power models for each component of the system. In research and industrial products different solutions for generating power models can be found. An excerpt of the wide range of tools is given in Section 2.3.4. In [99] power models are distinguished in three categories:

- datasheet-based power models
- measurement-based power models
- analytical models

Datasheet models are based on the information provided by the manufacturer of the device. Such models are often used for memory devices, as they have a defined set of operation modes. For SRAM they are simple, DRAM models are more complex, because the power consumption is access-dependent, i.e. it depends on previous accesses to the device. Measurement-based models rely on the measurement of the power consumption on real hardware devices. These models are very accurate, but the measurement might be difficult, since the supply voltage pins are often not easily accessible. The models are valid only for the measured device and cannot be transferred to other device. Analytical models are much more versatile, as they regard the inner structure of a component. If the component behavior is well modeled, the model is parameterizable and can be used for an entire device family. For example, analytical power models for memory devices can be configured in size and organization of data array. Such models might even be portable to other technology feature sizes, as for example the CACTI cache models. Analytical models are very complex and the accuracy might vary largely within the parameter range.

### 3.2.1 Processor Models

Processor models exist on different level of abstraction [78]:

- bus functional model
- instruction-level model
- cycle-accurate model

- nanosecond-accurate model

The bus functional model is the most basic description of a processor, which acts as black box and reflects only the interface behavior of the processor. Bus functional models (BFM) are often applied for testing hardware components in a hardware simulator, and serve the processors interaction with the hardware. This model is very fast, because only the interface to the hardware, usually a bus interface, is modeled. The models execution is controlled by a sequence of bus transactions, which are initiated or answered by processor. The simulation of software is not possible, because the processor internal structure, such as instruction decoding and execution is not simulated. Therefore the model is not feasible for profiling purposes.

Instruction-level modeling of processor allows the execution of software on the model. The executable implementation of such models, the instruction-level instruction set simulator (ISS), is the most widely spread model of processors. They are used for testing, profiling, co-simulating processors. For testing software the ISS is connected to a debugger, which controls the software execution. The GNU debugger gdb comes already with numerous built-in ISSes. The level of detail of the model can vary. At least the register file, the program counter and the status register are required for the model and the decoding and execution of the full instruction set must be ensured. More detailed models also model the pipeline, which ensures that pipeline stalls, data dependencies and branch delay slots are simulated correctly. The instruction-level ISS does not include timing information and therefore does not require detailed modeling of the memory subsystem, therefore access latencies and cache behavior is modeled. The correctness of memory accesses is often tested, including address validity and access alignment. Using instruction-level model a fast (almost real time) simulation can be accomplished and if the influence of the memory subsystem on the performance is low, this model can be used for efficient profiling. The gprof tool allows such profiling based on the results collected by gdb.

In order to increase the accuracy of the instruction-level models, it can be refined to resemble the real hardware behavior closer and annotated with timing information. Such models are used to build cycle-accurate ISSes, for example the ARMulator, which is described in Section 2.2.1 in more detail. The ISS must be accompanied by a timing model of the memory subsystem in order to allow realistic simulations. Cycle-accurate models are a good choice for profiling, as they reflect the timing behavior fairly accurate and provide a decent simulation speed.

The highest level of detail is implemented in nanosecond-accurate processor models. These models are described in hardware description languages, such as VHDL, Verilog or SystemC, and simulated in a hardware simulator. The simulation speed is extremely slow and only permits the profiling of short instruction sequences.

#### 3.2.1.1 Processor Power Models

Simulating the power of a processor can be based on datasheets, which leads to low complexity models. Datasheets only provide current or power consumption values for a few states, usually active, idle and a few types of standby modes. Simunic et al. [83] build a model that differentiates only between idle and busy states based on an instruction trace produced by an ISS. Tiwari et al. [97] present a more complex power model, which considers different power consumption for each instruction. The observation that similar instructions, i.e. instructions which use similar processor components, consume a comparable amount of power leads to a grouping of instructions, which simplifies the model. Additionally to the intra-instruction cost an inter-instruction cost is defined, based on the observation that the power consumption is dependent on the previously executed instruction.

Generally the following levels of accuracy can be defined, which reflect different influences:

- modeling only the processor state active/idle/standby (datasheet models)
- modeling of instruction groups, which activate similar functional units
- modeling each instruction separately
- modeling additionally each addressing mode separately
- modeling additionally the Hamming distance of coded instruction and data accesses
- modeling the influence of intra and inter-instruction cost ([97])

Besides these measurement-based models, analytical models exist, which reflect the processor architecture, which is based on simple building blocks, such as registers, adders and multiplexers. The power model summarizes the influence of these blocks depending on the activity of each block due to the processor behavior during a software simulation run. In Section 2.3.4.3 some models are presented, which are based on the analytical processor model SimpleScalar.

### 3.2.2 Memory Models

Memory has an enormous influence on both, the performance and the energy consumption of a system and should therefore be considered during the profiling. A high percent of the energy is consumed in caches, e.g. analysis of the StrongARM processor have shown that 43 % of the power is consumed in the caches [70], other studies [17] even state a percentage above 60.

**Caches** can be characterized by several parameters:

- size (in bytes) and line length
- associativity (direct-mapped, 2, 4, 8…, 64 sets or full-associative)
- replacement strategy (LRU, LFU, FIFO,…) and write strategy (write-through/write-back)
- special features, such as line-locking or pre-fetching

Cache timing simulators, such as Dinero IV [31] and Cheetah [93] apply these parameters for calculating the number of cache hits and misses. The simulation is trace driven, which means that the software is first executed on the target processor and the occurring memory accesses are traced and written to a file. This trace file acts as input to the cache simulator, which calculates the resulting number of misses and hits. The ARMulator comes with a built-in cache simulator, which besides the hit and miss calculation also allows a functional simulation, i.e. the full behavior of the caches is simulated including the data storage and transfer from the processor through the caches to the memory bus.

This information can then be used for determining the overall performance and energy consumption. The time and energy consumed by each cache miss and hit can be achieved with CACTI models [96]. CACTI takes the above-mentioned cache parameters and the technology feature size as an input, and calculates the timing, area and energy requirements for all cache components.

On-Chip memory components based on **SRAM** technology are usually running with the full processor speed and have single cycle access times. Therefore their timing behavior can be modeled easily by a static model for read or write accesses. Also the energy models are quite simple as the energy is not access dependent. [18] suggests using CACTI for SRAM models.

**SDRAM**, as described in Section 3.1.2.4, is far more complex than SRAM, which also leads to elaborated timing and power models of this memory. In order to have an accurate timing model, the entire control flow of the SDRAM controller needs to be reflected, which includes refresh, burst, page activation and standby modes. Some of these modes can be neglected without a major influence on the accuracy, such as the refresh cycles. The most influencing characteristic can also be described by a simpler timing model, which differentiates only between read and write operations and sequential and non-sequential access. The later is used to reflect the faster accesses in burst mode and in case of a page hit.

The simplest model for the power consumption assumes a constant value, which only depends on the clock frequency and the supply voltage. Thus the consumed energy only depends on execution time. A model which can be found quite often in literature is a datasheet-based model provided by a memory device manufacturer [59]. The model comes as a spreadsheet, which contains formulas that calculate based on a parameter set the power consumption for each SDRAM mode. The parameter set includes the clock frequency and supply voltage, and technology dependent timing and current parameters from the datasheet of the specific memory device. Furthermore some user-defined estimates for the memory activity are incorporated.

## 3.3 Profiling

This section gives an overview of profiling techniques and results. The software-centric profiling can take place on different levels of abstraction:

- source-code level (source-code analysis)
- instruction level (instruction-accurate ISS)
- cycle level (cycle-accurate ISS)
- exact timing (nanosecond-) level (HDL)
- hardware implementation level (FPGA, ASIC)

The level of abstraction also determines the accuracy of the results and mostly also the simulation time. During an early stage of the design, when only a software model of the system is available, an abstract source code analysis can take place, in order to get a first idea of the code complexity. If the decision for a specific processor is made, an ISS for the processor can provide instruction-accurate profiling results. If the ISS models the pipeline and the other processor components in detail, even cycle-accurate results are possible. For nanosecond-accurate profiling, hardware simulators are required.

Many profiling tools focus on performance measurements, as it is often the most important design goal, and provide cycle or execution time results. Besides this timing information, further profiling results may be very helpful during the design and optimization process. This includes memory profiling, such as read/write accesses, cache misses and bus utilization and also power profiling.

The profiling data can be assigned to the program code on different levels of abstraction. Besides the overall profiling for the entire software, the most common way is assigning the results to the functions in the code. More detailed results are assigned to each basic block, i.e. a sequential code segment without jump instructions. The highest accuracy is on assembly line level.

Besides the above described dynamic profiling, which is based on the code execution, a static profiling can be performed. Static profiling analyzes the program code and can be used for statistics on instruction usage, code size and performance estimation. The advantage is that the profiling is very fast and does not require any simulator or real hardware. However results about the actual behavior of the software can be only considered as a rough estimation.

### 3.3.1 Profiling Results

The most common result provided by profiling tools is the timing information about the software under test. Especially for software, which needs to fulfill strict real time constraints, this is the most important information. The timing is usually expressed in an SI time unit, e.g. milli- or nanoseconds, or in units of cycles corresponding to the processor clock frequency, such as core or external bus cycles. The chosen unit depends on the underlying hardware and software structure of the profiling environment. If the software is tested for a fixed hardware architecture with a specific clock frequency and timing of the hardware component, the timing can be given in the SI unit, which can be easily compared to the real time constraints of the system. If the software under test is running within a multi-tasking operating system, the timing results need to be separated in overall execution time and the actual time spent for the software, as some of the execution time is spent for other tasks or the operating system itself. Using multi-tasking in a system always carries the risk of producing erroneous profiling results, as the task switching has an influence in multiple unpredictable ways on the performance, for example the task switching may lead to cache trashing.

Additionally other statistical information can be of interest. The data access and transfer analysis is a crucial feature of profiling tools. This includes the register usage, cache, bus and memory activity. The cache profiling includes the number of cache hits, misses and fills and the bus activity provides an overview of bus cycles (e.g. read, write, burst, wait) and the peak and average transfer rates. The accesses to the memory can be split into memory areas, e.g. variables, heap or stack) and provide statistics on the kind of access and the required wait states. Besides real memory devices other peripheral components, such as memory-mapped I/O ports might be profiled. All these events have an influence on power consumption and performance, especially for data-intensive applications.

The profiler may also provide statistics on instruction execution. This includes an analysis of code coverage, i.e. which parts of the software are actually used during execution. The instructions set usage gives an overview of the assembly instruction, which are used during the execution of the code, and can be used to eliminate unused instructions from the ISA and for special functional units, such as floating point or SIMD instructions. The internal behavior of the processor might also be of interest, such as the success of branch prediction mechanism or number of stalls of the pipeline for studying their efficiency.

Besides the profiling of execution time, data accesses and processor states more and more tools arise [40, 17], which also consider the power or energy consumption, respectively. This knowledge support optimizations for energy efficiency, which is especially important for battery-powered devices.

### 3.3.2 Profiling Methods

Dynamic profiling is based on data collected during the execution of the software. The acquisition of runtime information about the program execution can be accomplished by different methods. The usage of these methods is also depended on the underlying software execution model.

#### 3.3.2.1 Code Instrumentation

This method inserts small code pieces into the source code of the program, which are used to create profiling information during the runtime. For example the gprof profiler inserts in each function code, to count how often the function was called, how much time was spent in the function, and which was the calling function for creating the callgraph. All this profiling information is written to a file and can be viewed with gprof. The advantage of code instrumentation is that no simulators are required, and that the execution speed is only decreased slightly; this leads to fast profiling results. Drawbacks are that this method is intrusive, i.e. the execution of the application is modified. This can lead to inaccuracy in the profiling results; especially for small functions the execution time of the instrumentation code can exceed the actual task performed in the function. Furthermore the instrumentation code is inserted at compile time, i.e. the source code is required for such profiling. Especially this inhibits profiling of third-party library, which are note provided with source-code. Also code instrumentation sometimes conflicts with compiler optimizations, which leads to profiling non-optimized code and lead to inaccuracy. The profiling results of this method are fairly restricted, as no hardware related information, such as cache misses or pipeline stalls, can be collected.

#### 3.3.2.2 Sampling

Sampling is a very widely used method for non-intrusive profiling. With this method, the execution of the program under test is accompanied by a profiling tool, which probes the program status at specific sample times. By probing the program counter at the sample point, the profiler can determine the current position in the code and update the profiling data of the current function or building block. Thus the number of times each function is called can be evaluated and the time spent in the function can be estimated. The sampling period is either timing or event-based, the former is triggered by an internal clock, usually in the range of milliseconds whereas the latter is triggered by processor events, such as cache misses. Since the probing only takes places at specific times, the profiler might “overlook” small functions, which are called between two sampling points. Especially for programs with a short execution time this leads to inaccurate and non-deterministic results. To overcome this problem, the program can be called in a loop, however this might lead to incorrect results, as the cache performance might be to optimistic, because code and data of the previously executed run are still in the caches. Usually the program code and the profiler are running on the real processor, such as with VTune. If the sampling point occurs, the profiler interrupts the processor, and collects the profiling data. The hardware needs specific features, in order to allow the interrupt and to provide the data. For example, if a cache miss occurs, the hardware needs to inform the profiling software about this event. Therefore the amount of profiling information differs for different processor types. The sampling-based profiling on real hardware allows fast and accurate profiling

#### 3.3.2.3 Profiling with Instruction Set Simulator

If an ISS, see Section 3.2.1, is available for the desired processor, it can be used for detailed and non-intrusive profiling. The software execution on the ISS is a few orders of magnitude slower than on real hardware, typically in the range a few hundreds of thousands to millions of cycles per seconds can be simulated. An internal view into the processor states is possible, such as register or cache accesses or pipeline stalls. By monitoring these events a detailed profile of the software execution can be created. This profiling is non-intrusive, because the software execution on the simulator is not effected by the monitoring. Many ISSes provide at least basic profiling abilities [9, 7, 30], for example the accumulated and self execution time

of each function or the pipeline or register usage. The profiler presented in this thesis is based on an ISS. Depending on the accuracy of the processor core model, usually either cycle- or instruction-accurate, the ISS provides more or less detailed information about the execution. The accuracy of the results is also influenced by the accuracy of the models of peripherals modules, such as cache, bus and memory components.

Some ISSes (e.g. the ARMulator) can be extended with hardware models of peripheral components of the system, such as coprocessors or IO devices. Such a hardware/software cosimulation environment allows a system profiling, which can be helpful during the partitioning and scheduling process of hardware/software systems.

#### 3.3.2.4 Hardware Simulator based Profiling

The most detailed profiling can be achieved with a hardware simulation of the processor. If a hardware description model of the processor, for example in VHDL or Verilog, exists, hardware simulators, such as Modelsim [69] allow a nanosecond-accurate simulation of the processor. The accuracy of the simulation depends on the granularity of the model, which is described on high-, register-transfer- or gate-level. Gate-level descriptions allow a 100 % accurate simulation of the processor and thus allow the highest accuracy for profiling of the software execution. Such profiling is however extremely slow, only a few to a few hundreds of cycles can be simulated per second, therefore it is not feasible for analyzing complex software. The processor can be extended with other hardware components and therefore also allows hardware/software cosimulation and profiling. The profiling with hardware simulators requires a monitoring extension in order to probe the internal states of the processor, for example by using the foreign language interface (FLI), as described here [90].

## 3.4 Data-Intensive Applications and their Implementation for RISC Processors

This work focuses on the analysis and optimization of data-intensive applications on embedded systems. Such applications, for example video players and recorders on mobile phones, have become very popular and make high demands on embedded systems. First of all, they are often very computationally intensive. Additionally, the huge amount of data transfers highly influences the performance and power dissipation and thus also the design of the system architecture. On the example of a highly complex application, an H.264/AVC video decoder will be used to show how the different algorithmic parts of the application are influencing the computational and data transfer requirements. Also the influence of the different types of memory access, sequential or random, its width, and access pattern should be considered.

This problem is even higher in RISC processor based embedded systems as compared to standard PC architectures, as their load/store architecture acts as the major bottleneck to the performance. Also the small register file, for example 16 registers for the ARM architecture and the small caches size increase this problem. The rising clock frequency used in embedded systems also increases the speed gap between CPU and memory. A problem, which until a few years ago was only apparent in PC architectures, now leads to the same high influence of the data transfers and memory architecture on the performance.

### 3.4.1 The H.264/AVC Video Coding Standard

H.264/AVC is the most recent video compression standard developed by the Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG [60]. Like its predecessors H.264/AVC uses a

block-based hybrid coding approach, which takes advantage of (motion compensated temporal and spatial) prediction and transformation of residual data. H.264/AVC adds various new coding features and refinements of existing mechanisms, which lead to a two to three times increased coding efficiency compared to MPEG-2. However, the computational demands and required data accesses have also increased significantly. Figure 12 shows the block diagram of an H.264/AVC decoder.

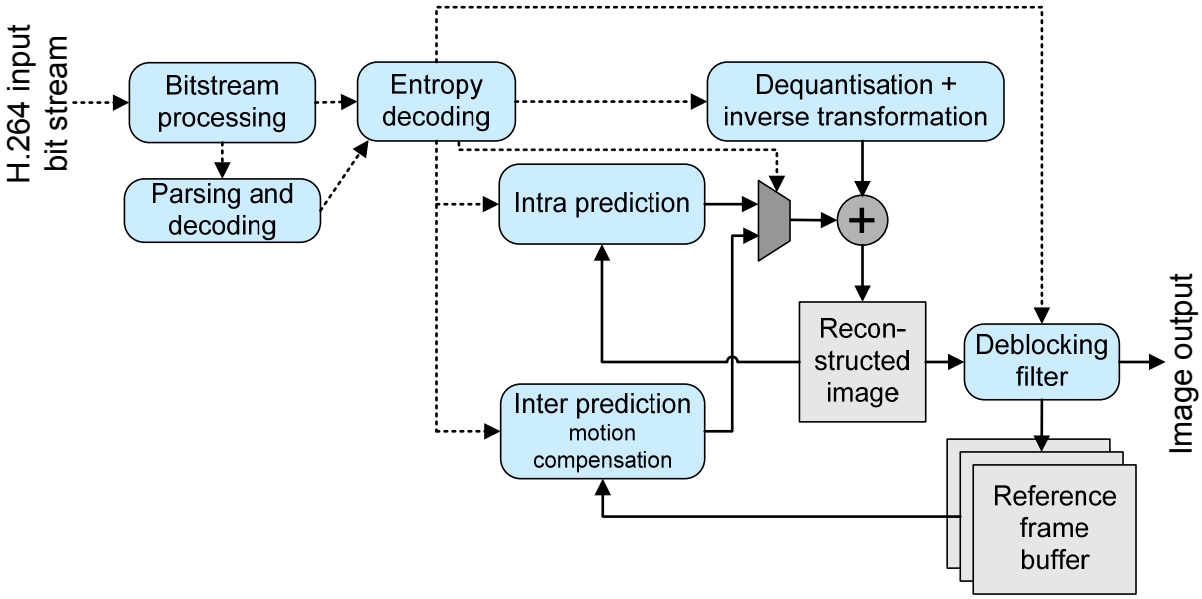



*Figure 12: Block diagram of an H.264/AVC decoder*

The decoder consists of five sequential computation steps, which are bitstream parsing, entropy decoding, prediction, coefficient transformation and deblocking. The bitstream processing unit parses the bitstream for symbols, which are then entropy decoded. H.264/AVC allows two different entropy coding modes, variable length coding (CAVLC) and binary arithmetic coding (CABAC). Both methods are context adaptive (CA-prefix), i.e. the coding parameters are adapted according to previous data in order to achieve a high compression. The decoded symbols contain control information, prediction data and transformed residual data. H.264/AVC provides inter and intra frame prediction modes to predict image data from previous frames or from neighboring blocks, respectively. Inter prediction can be performed on sub-macroblock level (down to 4x4 blocks) and the motion vector resolution goes down to quarter-pel precision requiring interpolation of pixel data. For intra prediction several modes are defined, e.g. horizontal prediction from the left neighboring macroblock or vertical prediction from left and upper neighbors. Intra prediction can either be performed on 16x16 or 4x4 blocks.

The residuals of the prediction are received as transformed and quantized coefficients. After inverse quantization and transformation of the coefficients the residuals are added to the predicted data, which leads to a reconstructed image. The transformation is performed on 4x4 blocks (in high profile 8x8 is also supported) and is based on integer arithmetic, contrary to the residual transformation in previous video compression standards applying a discrete cosine transformation (DCT). The reconstructed image is postprocessed by a deblocking filter for reducing blocking artifacts at block edges. The deblocked image is used for performing inter prediction whereas intra prediction is based on the reconstructed image.

For example, in H.264/AVC video decoding, half of the decoding time is spent with memory accesses. Figure 13 shows how the required memory accesses have risen from decoding MPEG-4 to decoding H.264/AVC.

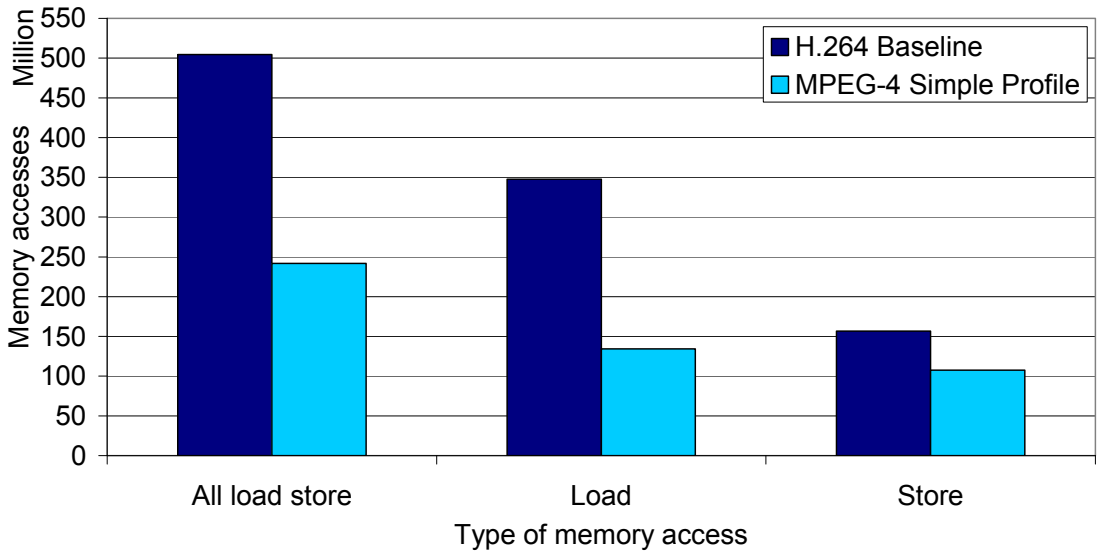



*Figure 13: Comparison of memory accesses required in MPEG-4 Simple Profile and H.264/AVC decoding*

A closer look to the algorithms of the functional parts should reveal which computational and data transfer demands occur. Figure 14 illustrates the mechanism behind the different parts. A simple inter prediction (grey box in Figure 14.a) implies only a copy operation of the block from the previous frame. This includes some address calculation and the two-dimensional byte copy. Almost the same applies if a motion vector is used with an integer number of pixels ("mv1"). If the motion vectors point to a non-integer position ("mv2") these sub-pixels need to be calculated by an interpolation mechanism. At first half-pixel (circles) are calculated as a weighted sum of neighboring full-pixels (crosses) and then in a second step quarter-pixel position (filled cycles) are linear interpolated from the neighboring full- and half-pixels. This is a demanding computational process. SIMD instructions might be applicable for the parallel filtering of adjacent pixels. The efficiency of caches is restricted due to the random manner of the motion vectors. Two-dimensional DMA transfers could be applied for integer motion vectors, however depending on the memory architecture they might be restricted due to address miss-alignment.

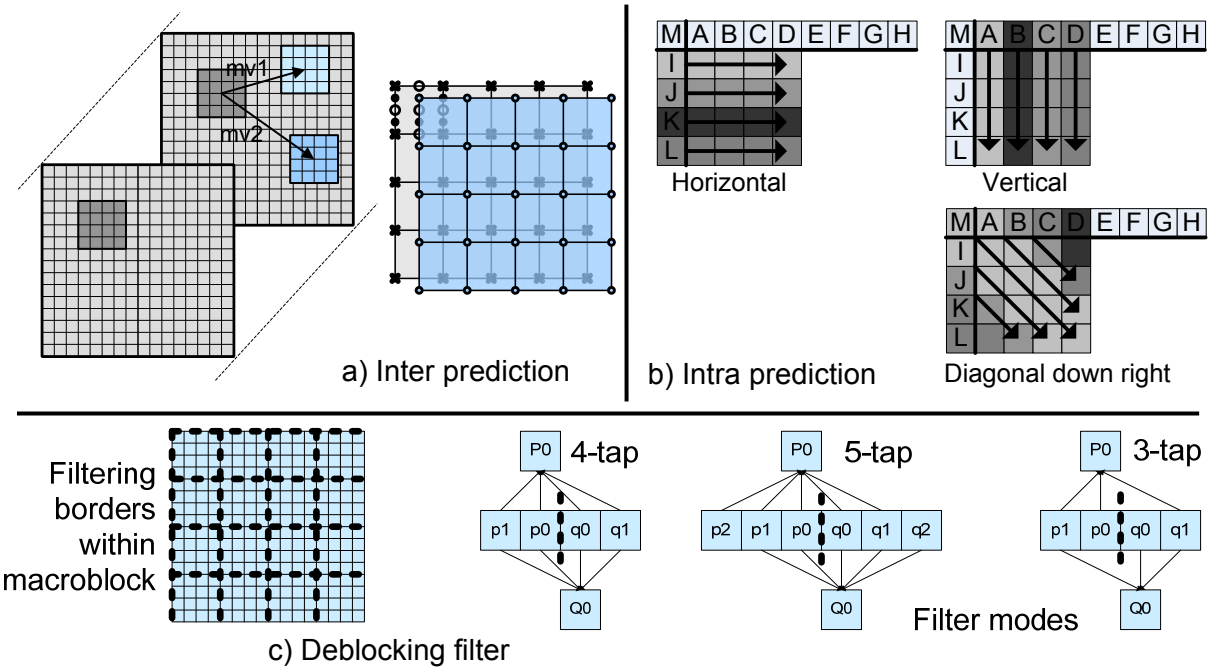



*Figure 14: Algorithmic parts of H.264/AVC video coding*

Intra prediction is a similar approach of reusing pixels, but here pixels are taken from the spatial neighborhood. Figure 14.b) illustrates three of the nine available modes, where left and upper neighboring pixels are used for prediction. In simple modes the pixels are just copied, more advanced modes interpolate between several neighbors. Again for simple modes DMA modes might be applied, here in a one- to two-dimensional manner, and SIMD instruction might be suitable for parallel processing of multiple rows or columns. Depending on the processing order, caches should increase the performance, because neighboring blocks are usually processed sequential order, and therefore the pixels should be still available in the cache.

The deblocking filter, shown in Figure 14.c) applies filters with various tap-lengths to the borders of each 4x4 pixels block. This makes the deblocking very computationally demanding. The processing is performed on a macroblock basis, first for vertical than for horizontal borders. Thus the processing is very local and data transfer can be accomplished efficiently with two-dimensional DMA transfers for the entire macroblock. Due to this data locality, caches should also work efficiently. SIMD instructions might be used on a 4x4 block basis.

The inverse integer transformation is applied on 4x4 or 8x8 block basis. Again accesses are spatially restricted and should benefit from caches. The usage of DMA transfers is highly dependent on the arrangement of the incoming coefficient data, as they are usually ordered in a sequential, one-dimensional manner when they are read from the bit-stream. SIMD instructions should be suitable, as the transformation applies similar calculations for each row and column.

The two entropy coding mechanisms, CABAC and VLC, are more control than computationally intensive. The entropy coding is based on the mechanism to adapt the binary representation of information to the probability of the symbols to be transmitted, i.e. short code words are used for often used symbols. VLC uses tables for the mapping of the symbols to the code words. These tables can be either implemented as look-up tables or arithmetically described. The access to these tables is random, therefore caches might not work efficiently and these tables can be stored in fast memory. An arithmetical implementation is more computationally demanding but has the advantage of less memory requirements and accesses.

CABAC is a more complex coding technique based on arithmetic coding. Contrary to VLC CABAC performs a continuous coding of the information instead of coding each symbol separately. This allows a higher compression ratio, however it is computationally intensive and the dependency between the coding steps inhibit parallelization or pipelining.

# 4 Comprehensive Profiling of Embedded Processors

This chapter describes a methodology developed for extensive profiling and demonstrates its application for software optimization and architecture design of embedded systems. The work focuses on analyzing memory accesses and power consumption of processors and their accompanying memory architectures. The first section gives an overview of the requirements for such analysis and describes the method applied to achieve the desired results.

The following sections present numerous techniques that have been developed in order to optimize the memory-related issues of the software and hardware parts of embedded systems. The sections also show how the profiling results can be utilized for this purpose.

In order to incorporate power consumption estimation in the profiling process, a power model has been developed, which reflects the processor and the surrounding on-chip components. Section 4.5 describes the measurement techniques and the development of a model based on measurement of an SoC. A software test suite is shown, which has been generated for extracting the different influences that contribute to the power model.

## 4.1 Extensive Profiling Methodology

The most common result delivered by profilers is the instruction cycle count information. In the case of data-intensive and complex applications this information is not sufficient, as the influence of the memory and system architecture needs to be taken into account. Therefore, for these applications an analysis method is required, which needs to address the following aspects:

- fast simulation time, as complex applications require the analysis of a long instructions sequence to achieve significant results
- detailed and accurate results for finding the hot spots in the software und revealing the reason for a performance or power consumption issue
- accuracy concerning the hardware architecture, as the memory and bus timing influences the overall performance significantly
- gather memory access statistics, in order to pinpoint the influence of the memory accesses within the software

For many decisions in embedded system design, more detailed information about memory accesses is required, as these accesses have an enormous influence on performance and power consumption. The following list shows a selection of useful parameters:

- clock cycles budget
- details for memory accesses, with access direction (load or store) and size (bit-width)
- cache activity, such as hit and miss ratio
- bus activity, such as workload and bus master assignment
- access statistics for variables and specific memory areas
- distribution of executed instructions
- power consumption

These profiling results can be very useful for both hardware and software optimizations. However, especially for software modification, the place within the software which produces the specific event needs to be identified. Therefore the results need to be on a fine-grained level, for example function or basic block level. On the other hand, if the application, or parts of it, are only available as libraries or object code, an analysis should be still possible. Thus the profiling should not be dependent on the source code.

Each of the tools presented in Section 2.3 fulfills only a sub-set of these requirements. Therefore a profiling method has been developed and implemented that provides the required analysis results.

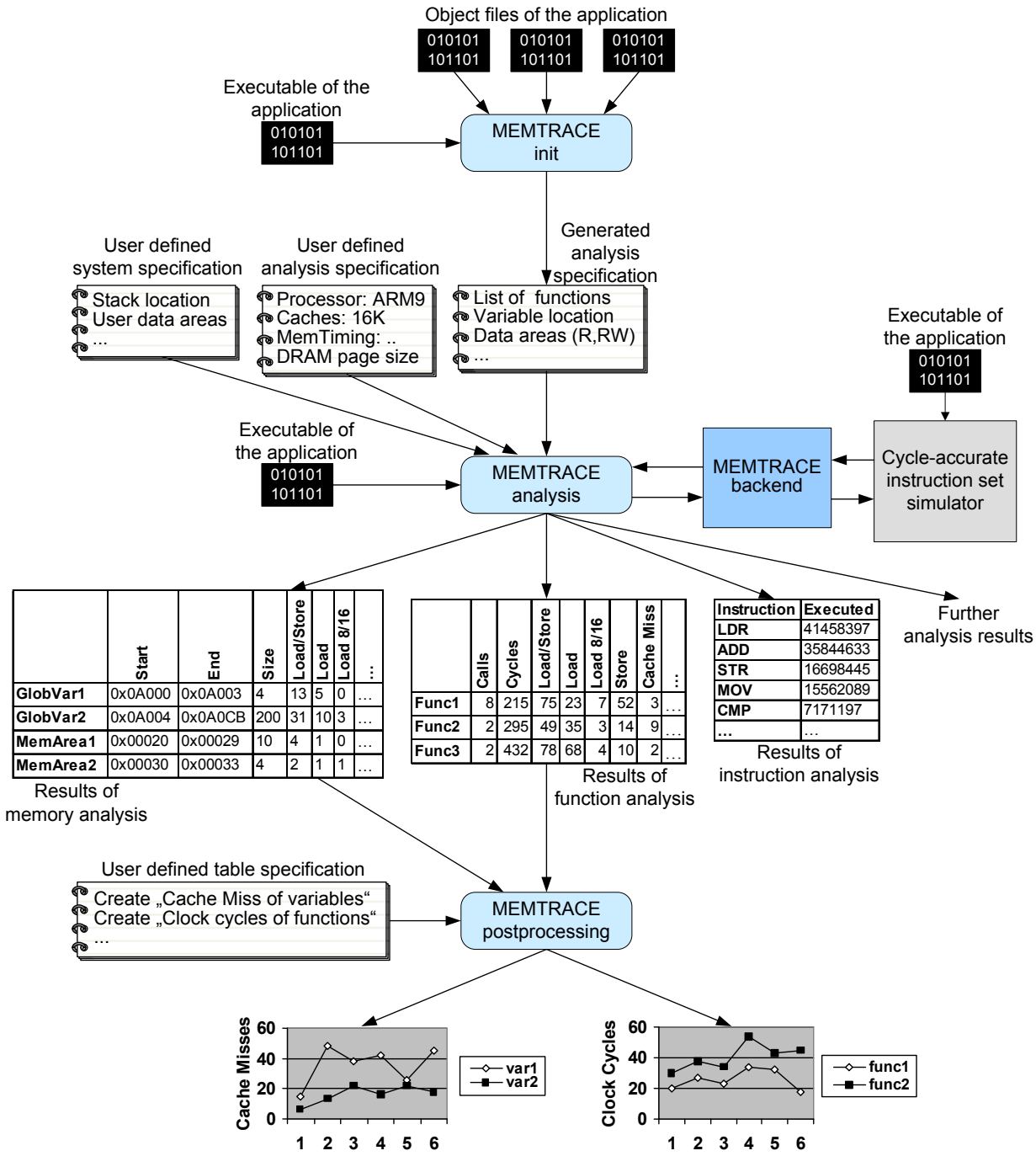
Considering the requirements described above, an ISS-based profiling as described in Section 3.3.2.3, has been found to be the most appropriate choice for this purpose. It allows a fast simulation and a comprehensive view inside the processor hardware and often provides multiple options for architecture adjustments. The accuracy of the ISS should be on a cycle level, in order to be sufficient for the optimization and architecture decisions to be made.

The performance analysis developed in this work is carried out in three steps: the acquisition of program information, the acquisition of profiling data during the runtime of the program and the representation and postprocessing of the results. This toolflow is described in more detail in the following sections. It has been implemented as the MEMTRACE profiling tool, shown in Figure 15.

The acquisition of profiling data is performed by connecting the tool to an ISS and gathering the information provided. A tracing-based method is used for this purpose, which traces the following basic information:

- program counter
- cycle counter
- data and instruction busses including their address busses
- optional further information, such as cache misses or external bus usage

In Figure 16 the interconnection with the ISS is shown, including the tracing probes on the instruction and data bus and the external system bus. The program counter defines the current position within the program code. In conjunction with the cycles counter, the distribution of the total execution time over the executed assembly code can be determined.




| | Start | End | Size | Load/Store | Load | Load 8/16 | ... |
|---|---|---|---|---|---|---|---|
| GlobVar1 | 0x0A000 | 0x0A003 | 4 | 13 | 5 | 0 | ... |
| GlobVar2 | 0x0A004 | 0x0A0CB | 200 | 31 | 10 | 3 | ... |
| MemArea1 | 0x00020 | 0x00029 | 10 | 4 | 1 | 0 | ... |
| MemArea2 | 0x00030 | 0x00033 | 4 | 2 | 1 | 1 | ... |

| | Calls | Cycles | Load/Store | Load | Load 8/16 | Store | Cache Miss | ... |
|---|---|---|---|---|---|---|---|---|
| Func1 | 8 | 215 | 75 | 23 | 7 | 52 | 3 | ... |
| Func2 | 2 | 295 | 49 | 35 | 3 | 14 | 9 | ... |
| Func3 | 2 | 432 | 78 | 68 | 4 | 10 | 2 | ... |

| Instruction | Executed |
|---|---|
| LDR | 41458397 |
| ADD | 35844633 |
| STR | 16698445 |
| MOV | 15562089 |
| CMP | 7171197 |
| ... | ... |

*Figure 15: The MEMTRACE toolflow: 1) The init step extracts the function and data area names of the application to be analyzed. 2) The analysis step takes this list of names and a number of user-defined parameters for controlling the actual analysis. An ISS is used for running the application and the MEMTRACE backend performs the runtime data acquisition. The outcomes of this step are the different profiling data results. 3) The postprocessing step extracts user-defined data from the profiling data and creates spreadsheet tables for further statistical analysis.*

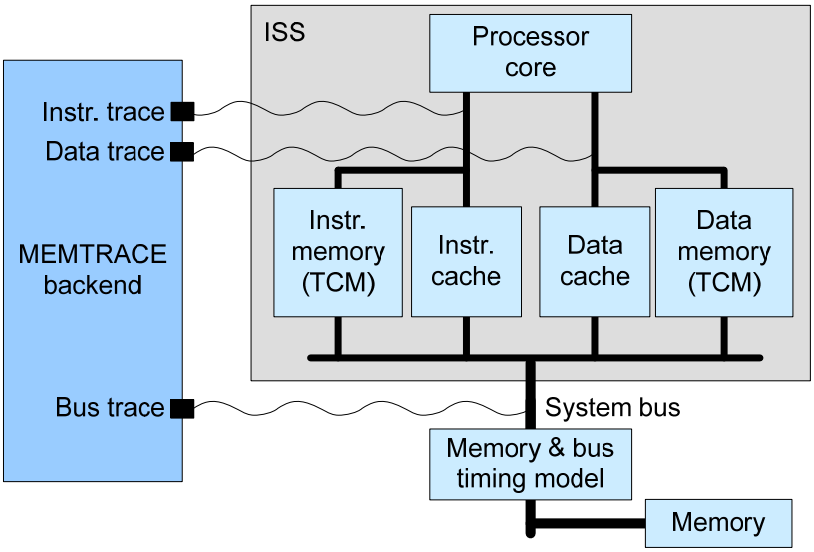



*Figure 16: Backend connection to the instruction set simulator*

In order to support an evaluation on a C source code function granularity, the assembly lines need to be mapped to the corresponding functions. The C compiler creates labels in the assembly code, which identify the beginning of a function. The label addresses are stored in the symbol table of the executable code. These labels can be applied for the mapping of the functions. The address range of a function is defined as:

*start address = label address*

*end address = next label address - one word*

Figure 17 shows an example of address mapping for three functions, which are located at consecutive addresses in the instruction memory.

| Label | | Address |
|---|---|---|
| | ... | |
| | ... | |
| func1 | PUSH {r4-r6,r14} | 0x8f70 |
| | MOV r6,r2 | |
| | MOV r5,r0 | |
| | B 0x8f98 | |
| | LDR r2,[r5,#0x1c] | |
| | LDRB r0,[r4],#1 | |
| | LDR r1,[r5,#0x24] | |
| | MOV r14,pc | |
| func2 | LDR r1,[r0,#0] | 0x8f90 |
| | ADD r2,r1,#1 | |
| | STR r2,[r0,#0] | |
| | LDRB r0,[r1,#0] | |
| | MOV pc,r14 | |
| func3 | PUSH {r5-r8,r14} | 0x8FA4 |
| | MOVS r5,r1 | |
| | PUSH {r4} | |
| | MOV r4,r0 | |
| | ADD r7,r0,#0x3c | |
| | ... | |
| | ... | |

func1:

Start address = 0x8f70

End address = 9x8f8C

func2:

Start address = 0x8f90

End address = 9x8fA0

func3:

Start address = 0x8FA4

End address = …

*Figure 17: Mapping the instruction memory address range to C source code functions*

Similarly, the address of global variables can be extracted, as their addresses and size are also given in the table. As the location of local and heap variables are generated during the run time, they can not be extracted from the code. However, instead the stack and heap can be monitored, if their location and size is known. If the source code is written in C, the labels usually correspond to the function name. C++ function names are encoded differently; the symbol for a function combines the namespace, class, member function name and parameter types. The mangling is compiler dependent. In order to extract the function names tools such as c++filt [35] can be used.

To achieve a more detailed level of profiling than the function level, the source code lines can be mapped to the assembly code addresses. A table for such a mapping is available in the executable, if the compiler is instructed to enclose debug information. With gcc, the option -g is used for this purpose.

### 4.1.1 Program Information Acquisition

During initialization, the names of all functions and variables of the application are extracted. During this process, user variables and functions are separated from standard library functions, such as printf() or malloc(). This is achieved by comparing the symbol table of the executable with the ones belonging to the user library and object files. The results are gathered as the analysis specification. This specification can be modified, e.g. for adding memory areas, such as the stack and heap variables, for additional analysis. In order to generate intermediate results, for example to generate separate profiling results each time a specific function is called, this function is tagged with “split”. The profiler is instructed to produce snapshot results, each time the "split function" is called and to reset the profiling counters. Additionally, the analysis specification controls whether the results, e.g. clock cycles, of a function should include the results of a called function (accumulated) or if it should only reflect the function’s own results (self). Typically, auxiliary functions, e.g. C standard library or simple arithmetic functions, are added to the calling functions. The system specification provides information on the processor type and the memory architecture, including cache size, page size and memory timing.

### 4.1.2 Runtime Data Acquisition

In the second step the performance analysis is carried out, based on the analysis specification and the system specification, as shown in Figure 15. The system specification includes the processor, cache and memory type definitions. The ISS is applied to simulate the user application and the profiler gathers the analysis results of the functions and variables. Section 5.2.2 describes this process in more detail. Table 3 shows example results for function profiling. The output files serve as a database for the third step, in which user-defined data are extracted from these tables.

MEMTRACE communicates with the ISS via its backend, as depicted in Figure 16. Initially, the backend creates a list of all functions and marks the user and split functions as defined in the analysis specification. For each function a data structure is created, which contains the function’s start address and variables for collecting the analysis results. To accumulate results of called functions to the calling function, the analysis uses two tags, `currentFunction` and `evaluatedFunction`, to identify these functions. The former indicates the function currently being executed. The second tag is used when this function should not be evaluated. Then the second tag indicates the calling function, to which the result of the current function should be added.

Each time the program counter changes, MEMTRACE checks if the program execution has changed from one function to another. If so, the cycle count of the `evaluatedFunction` is recalculated and the call count of the `currentFunction` is incremented. Finally, the pointers to the `currentFunction` and `evaluatedFunction` are updated. If the `currentFunction` is a split function, then the differential results from the last call of this function up to the current call are printed to the result files.

For each access that occurs on the data bus (to the data cache or TCM), the memory access counters of the `evaluatedFunction` are incremented. Depending on the information provided by the ISS, it is decided if a load or store access was performed, and which bit-width (8/16 or 32 bit) was used. Furthermore, the ISS indicates if a cache miss occurred. Page hits and misses are calculated by comparing the address of the current memory access with the previous one and incorporating the page structure of the memory.

For each bus cycle (on the external memory bus) MEMTRACE checks if it was an idle cycle, a core access or DMA access, and increments the appropriate counter of the `evaluatedFunction`.

At the end of the simulation, the results of the last `evaluatedFunction` are updated and the results of the last call of the split function as well as the accumulated results are printed to the result files.

### 4.1.3 Representation of the Statistical Analysis Data

Table 3 shows that for each function, numerous profiling results are provided. In the first column the number of calls of the function is given, followed by the exact number of clock cycles spent in this function. The cycle count refers to the system bus speed, which might differ by a specific factor from the processor core cycles. To achieve the core cycles, the given cycles need to be multiplied by the factor. Furthermore, memory access statistics are given. All load and store operations are summed up, data as well as instruction accesses. These are accesses initiated from the load/store interface of the processor core, see Figure 16. Thus if the core is equipped with caches or MMUs, the load and store operations are accesses to those components; otherwise they reflect the accesses as visible on the memory bus. The accesses are also provided separately as load and store accesses, as well as according to their data width. The number of short (8/16 bit) accesses is also given.

*Table 3: Example table of function analysis results*

| Function | Calls | Cycles | Load/store | Load | Load 8/16 bit | Store | Store 8/16 bit | Page miss | Cache miss | Bus CPU | Bus DMA | Bus ille |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Func1** | 8 | 215 | 75 | 23 | 7 | 52 | 3 | 42 | 5 | 123 | 92 | 0 |
| **Func2** | 2 | 295 | 49 | 35 | 3 | 14 | 9 | 17 | 9 | 55 | 153 | 87 |
| **Func3** | 2 | 432 | 78 | 68 | 4 | 10 | 2 | 31 | 17 | 143 | 289 | 0 |

To reflect the influence of the page structure of dynamic memory, the profiler performs simple page emulation. The results for page misses and hits are given. If the processor is equipped with caches, the number of data cache misses is also given. To inspect the bus activity, the (system bus) clock cycles are separated into CPU, DMA and idle cycles, with the cycle type

reflecting the initiator of a transfer (or busy) bus cycle. The profiling results for memory areas and variables (Table 4) are similar to the function result. Additionally, for each memory area the start and end address and the size are given.

*Table 4: Example table of memory analysis results*

| | Start address | End address | Size | Load /store | Load | Load 8/16 bit | Store | Store 8/16 bit | Page hit | Page miss | Cache miss |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **GlobVar1** | 0x0A000 | 0x0A003 | 4 | 13 | 5 | 0 | 7 | 2 | 2 | 11 | 2 |
| **GlobVar2** | 0x0A004 | 0x0A0CB | 200 | 31 | 10 | 3 | 21 | 6 | 4 | 27 | 8 |
| **GlobVar3** | 0x0A0CC | 0x0A0D8 | 12 | 16 | 16 | 0 | 0 | 0 | 12 | 4 | 4 |
| **MemArea1** | 0x00020 | 0x00029 | 10 | 4 | 1 | 0 | 0 | 3 | 2 | 2 | 2 |
| **MemArea2** | 0x00030 | 0x00033 | 4 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

These comprehensive profiling results may lead to a huge amount of data, which can be hard to review. In order to allow the extraction of meaningful information from these statistics, a postprocessing of the data is required. This is performed in the third step, as depicted in Figure 15. MEMTRACE allows the generation of user-defined result tables from the performance results. Results of several functions can be accumulated into user-defined groups and are sorted by different criteria, e.g. for comparing the results of one group or for comparing one specific result for all groups. The tables can be further processed by spreadsheet programs, such as Microsoft Excel, e.g. for creating diagrams of the results.

Besides this profiling data, the tool provides further statistics, which are discussed in Sections 4.3 and 4.4. This includes an overview of successive accesses to neighboring pixels, which can be used to optimize memory accesses. To optimize the register usage, the memory locations and corresponding source code line are given for locations which are frequently accessed. The results for instruction and address mode profiling give an overview of usage of these architectural features, which is useful in cases where the instruction set of the RISC core should be adapted.

## 4.2 Memory Profiling within the Design Flow

This section describes how profiling can be applied during the design of embedded systems. As surveyed in Section 2.1.1, throughout the entire design flow, system analysis has a crucial influence on the performance and efficiency of the design. The following sections cover methods for optimization and exploration for all steps of the design process.

### 4.2.1 Hardware/Software Partitioning and Design Space Exploration

In order to define a starting point of a system architecture, an initial design space exploration should be performed. These steps include a variation of the following parameters:

- processor type
- cache size and organization
- tightly coupled memory
- bus timing

- external memory system and timing (DRAM, SRAM)
- hardware accelerators, DMA controller

The ability to configure these parameters easily between several profiling runs is crucial for testing the influence of the system architecture on the performance. These initial profiling runs also reveal the hot spots of the software. The most time-consuming functions are good candidates for either software optimization or hardware acceleration. Computationally intensive functions are especially well-suited for hardware acceleration in a coprocessor. With the support of a DMA controller, even the burden of data transfers can be taken from the processor. Control-intensive functions are better suited for software implementation, as a hardware implementation would lead to a complex state machine, which requires long design time and often doesn't allow parallelization. In order to get an initial sense of the influence of hardware acceleration, a factor (based on a well-educated guess) can be defined for each hardware candidate function. MEMTRACE uses this factor to manipulate the original profiling results.

### 4.2.2 Software Profiling and Optimization

After a partitioning in hardware and software is found, the software part can be optimized. Numerous techniques exist for optimizing software, such as loop unrolling, loop invariant code motion, common sub-expression elimination or constant folding and propagation. For computationally intensive parts, arithmetic optimizations or SIMD instructions can be applied, if such instructions are available in the processor. If the performance of the code is significantly influenced by memory accesses, as is generally the case for video applications, the number of accesses has to be either reduced or accelerated. The profiler gives a detailed overview of the memory accesses and thus allows identification of their influence. Based on this information, the optimization technique described in Section 4.3 can be applied.

### 4.2.3 Hardware/Software Profiling and Scheduling

Besides the software profiling and optimization, a system simulation including the hardware accelerators needs to be carried out in order to evaluate the overall performance. Usually hardware components are developed in a hardware description language (HDL) and tested with an HDL simulator. This task requires long development and simulation times. Therefore HDL modeling is not suitable for the early design cycles, where exhaustive testing of different design alternatives is important. Furthermore, if the system performance is data-dependent, a large set of input data should also be tested to get reliable profiling results. Therefore, a simulation and profiling environment is required, which allows short modification and simulation time.

For this purpose, an ISS can be extended with simulators for the hardware components of the system. For example the ARMulator ISS, see Section 2.2.1, provides an extension interface, which allows the definition of a system bus and peripheral bus components. It comes with a bus simulator, which reflects the industry standard AMBA bus. The simulator incorporates a timing model for access times to memory-mapped bus components, such as memory devices and peripheral modules. Figure 18 shows an example simulator setup for an embedded system containing a processor with a DMA controller, coprocessor and two memory components.

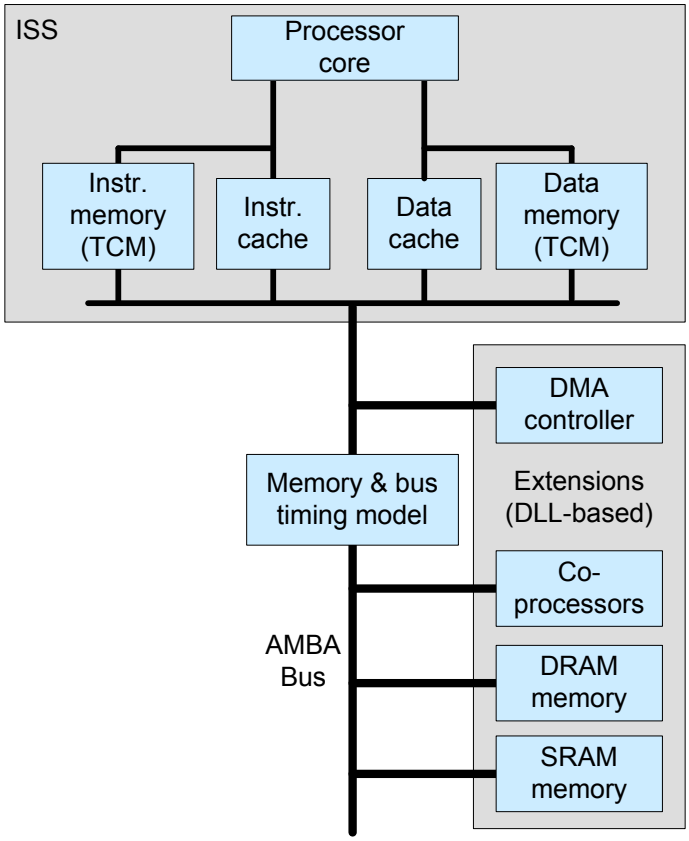



*Figure 18: Environment for hardware/software cosimulation and profiling*

### 4.2.4 Coprocessors

The system has been supplemented with a simple template for coprocessors, including local registers, memory and a cycle-accurate timing. The functionality of the coprocessor can be defined as C source code, thus the software function can be simulated as a hardware accelerator by copying the software code to the coprocessor template. The timing parameter can be used to define the delay of the coprocessor between activation and result availability, i.e. the execution time of the task as it would be in real hardware. This value can be achieved either from reference implementation found in literature or by an educated guess by a hardware engineer. Furthermore, often multiple hardware implementations of a task with different execution time (and hardware cost) are possible. In the proposed profiling environment, simply by varying the timing parameter and viewing its influence on the overall performance, a good trade-off between hardware cost and speedup can be found quickly. The bus interface of the coprocessors is described in more detail in Section 5.6.1.

### 4.2.5 Scheduling

After the software and hardware tasks have been defined, a scheduling of these tasks is required. To increase the overall performance, a high degree of parallelization should be accomplished between hardware and software tasks. In order to find an appropriate scheduling for parallel tasks, the following information is required:

- dependencies between tasks
- the execution time of each task

- data transfer overhead

Especially for data-intensive applications, the overhead for data transfers can have an enormous influence on performance. The speedup of a hardware accelerator might even be cancelled out by the overhead for transferring data to and from the accelerator.

The overhead for data transfers to the coprocessors is dependent on the bus usage. Furthermore, side effects on other functions may occur if bus congestion occurs or when cache flushing is required in order to ensure cache coherency. In order to find these side-effects, detailed profiling of the system performance and the bus usage is necessary. MEMTRACE provides these results; for example Figure 19 shows the bus usage for each function depending on the access time of the memory.

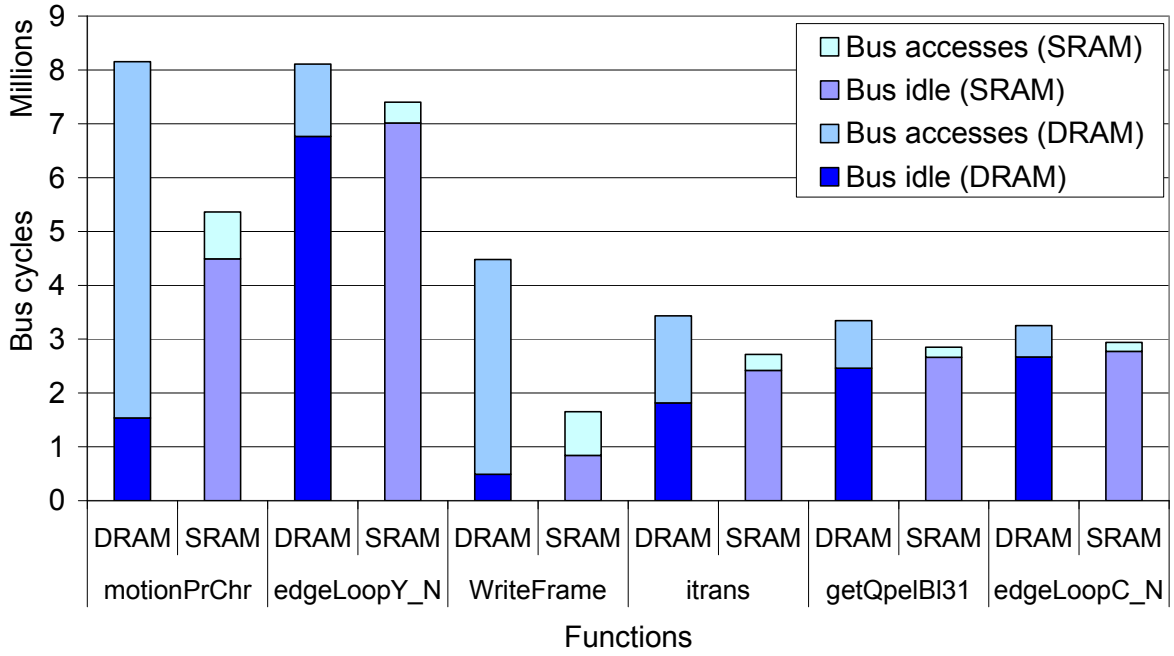



*Figure 19: Bus usage for each function, depending on the memory type*

### 4.2.6 HDL Simulation

In a later design phase, when the hardware/software partitioning is fixed and an appropriate system architecture has been found, the hardware component needs to be developed in a hardware description language and tested using an HDL simulator, such as Modelsim. Finally, the entire system needs to be verified, including hardware and software components. For this purpose, the ISS and the HDL simulator have to be connected. The codesign environment PeaCE [44] allows the connection of the Modelsim simulator and the ARMulator.

## 4.3 Profiling-Based Software Optimizations

Optimizing the hardware and software of an embedded system includes not only generic software optimizations [12], but also custom-tailored solutions for the specific application and the system architecture. Profiling supports the identification of hot spots in the application, which require optimization. The comprehensiveness of the profiling results also facilitates the decision as to what kind of optimization is appropriate, and thus the designer is aided during the optimization process. The optimization steps presented here are very much dependent on

the underlying system architecture. Therefore, the analysis results can be helpful especially when reusing software that has been written for other processor architectures or without a focus on speed optimization. Thus, this optimization methodology increases the portability and reusability of source code. The most important profiling results for optimization are cycle counts. They allow identification of the most demanding part of the software. General observations of the execution time of software have shown that 80 % to 90 % of the execution time is spent in 10 % to 20 % of the code. This rule, which follows the Pareto principle, leads to the dictum "make the common case fast". Amdahl's law [2]

$$overall\ speedup = \frac{1}{(1-P) + \frac{P}{S}} \tag{3}$$

*S: Speedup of code that is optimized*

*P: Portion of **executed** code that is optimized*

describes the speedup that can be achieved by optimizing these parts. For example, if a part of the software can be found where 50 % of the computations take place (P = 0.5) and the speed of this part can be increased by a factor of 2, than this would lead to an overall speedup of 33 %:

$$overall\ speedup = \frac{1}{(1-0.5) + \frac{0.5}{2}} = 1.33 \tag{4}$$

Contrary to this result, if the speed of a part of the software with 20 % of the computations is increased by a factor of 10, this only leads to a 22 % performance increase. This shows that if a hot spot can be found, this is the most promising candidate for high optimization potential. In the first step, the general software and hardware optimizations can be considered, e.g. loop unrolling, the usage of SIMD instructions or adjusting the cache size or memory architecture. After each optimization step, a new profiling run can be used to evaluate its influence. This is important, as the supposed optimization might interfere with other parts of the system and lead to reduced performance. Whereas shallower profiling would only show the overall influence, the detailed results help to find the cause of such interference. Besides these optimizations, in the following sections some mechanisms are shown which especially benefit from the memory access statistics provided by the profiler.

The following optimizations are often very specific to the actual compilation run and software version; therefore they should be applied in a very late step of the design. A recompilation might lead to data and program code placement in memory being changed, and thus to modified cache and bus usage and page miss behavior. Thus previous optimization steps, for example for data placement, might lead to worse results than without the optimization. The compiler needs to be instructed to take these modifications into account. Section 2.3.1 shows an example of how to create fixed memory maps.

### 4.3.1 Pinpointing Code Locations with Inefficient Memory Accesses

Many multimedia applications work on data with a size of a byte or a half-word. However, the memory bus width in many embedded systems is larger, e.g. 32 bits. Thus the available bandwidth is not used efficiently. To increase system performance, the entire bus width should be used and therefore byte or half-word accesses should be combined to word accesses. This is possible if adjacent address positions are accessed within a short time period. Figure 20

shows an example of memory accesses to nine adjacent bytes. The different offsets show the four different possible positions of the accesses in relation to word-aligned addresses.

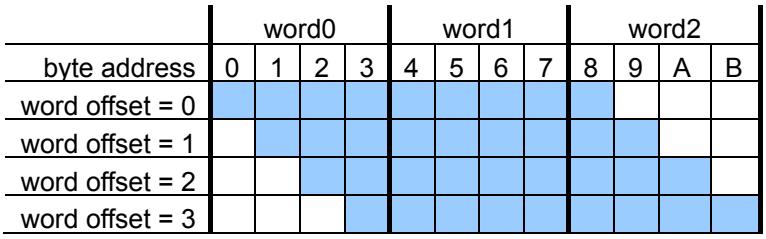



*Figure 20: Merging nine adjacent byte accesses to three 32-bit word accesses starting at any word offset position*

The figures shows that for any word offset position, it is sufficient to read three 32-bit words (word0 to word2) from the main memory. This corresponds to a three times reduction in the number of accesses. However, when reading multiple bytes at once, processing each byte requires shifting and masking operations, which leads to a computational overhead.

Listing 9 shows a code example for reading four adjacent bytes, the one at the current address (R0), the two on the left side (L0, L1) and the one on the right (R1). The if case shows how the byte-to-word conversion is implemented for a word offset of three. Cases for the other offsets are similar.

```
unsigned int w1,w2;
unsigned char L1,L0,R0,R1;
if (((unsigned int)SrcPtr&0x3)==0) {
  w1 = *((unsigned int*)((unsigned int)SrcPtr)-4); // left word
  w2 = *((unsigned int*)((unsigned int)SrcPtr));   // right word
  L1  = (w1>>16)&0xFF ; // extract 2nd left byte from left word
  L0  = (w1>>24)&0xFF ; // extract 1st left byte from left word
  R0  = w2&0xFF ;       // extract 1st right byte from right word
  R1  = (w2>>8)&0xFF ;  // extract 2nd right byte from right word
}
else { // fallback case for any access with word offset != 3
  L1  = SrcPtr[-inc2] ;
  L0  = SrcPtr[-inc ] ;
  R0  = SrcPtr[    0] ;
  R1  = SrcPtr[ inc ] ;
}
```

*Listing 9: Word access for adjacent bytes for a pixel address with word offset = 3*

Converting byte accesses to word accesses only speeds up the design if the overhead for shifting and masking is less than the time saved due to the reduced number of memory accesses. Furthermore, a processor with a data cache generally does not benefit from the byte-to-word conversion, as data of adjacent pixels is available in the cache due to the arrangement of cache lines.

Applying this optimization step requires the knowledge of the location in the code where such accesses occur. Finding these locations manually can be difficult or even impossible, especially if an advanced multi-step address calculation is performed. For example, in nested loops where multiple loop-parameters are incorporated in the address generation, the resulting address is not obvious. In this case, profiling the memory access pattern helps to find these locations. MEMTRACE provides memory access results for each function and differentiates between the bit-width of the access, as shown in the row “before optimization” in Table 5. As can be seen, about 30 % of the overall load operations are byte and half-word accesses. This

high fraction of non-word accesses provides a hint that this function might offer potential for byte-to-word conversion in memory accesses. However, the tool does not provide the information necessary to meet the second condition, which is that the accesses need to be to adjacent addresses. Nonetheless, it supports the designer by highlighting potential candidate functions for optimization. In this case, the function allows such as conversion, reducing the overall memory accesses by about one third and the execution of the function by more than 28 %.

*Table 5: Analysis results for a function (motCompChroma()) of the H.264/AVC decoder*

| | Clock cycles | All load | Load 8/16 |
|---|---|---|---|
| **Before optimization** | 13149109 | 309368 | 104784 |
| **After optimization** | 9355709 | 196746 | 34584 |

For more advanced information, MEMTRACE also provides a detailed list of source code line locations where such sequential accesses occur. Successive memory accesses are traced for this purpose and the distance between their addresses is calculated, thus neighboring addresses can be identified. The results are provided in a table, as in Table 6.

*Table 6: Location information about successive load and store operations to neighboring addresses*

| Number of loads | Number of stores | Function | File | Line | Assembly address |
|---|---|---|---|---|---|
| 0 | 52 | getNextVidAUH264 | testvidec.c | 347.2 | 8888 |
| 0 | 52 | getNextVidAUH264 | testvidec.c | 348.2 | 888C |
| 0 | 1371 | getNextVidAUH264 | testvidec.c | 349.2 | 8894 |
| 0 | 106 | getNextVidAUH264 | testvidec.c | 359.11 | 88C0 |
| 0 | 404 | DecodeH264 | decodeH264.c | 144.2 | 9474 |
| 0 | 4 | edgeLoopY_S | deblocking.c | 1125.14 | FA2C |
| 0 | 40 | edgeLoopY_S | deblocking.c | 1130.14 | FA64 |
| 0 | 1473 | edgeLoopY_S | deblocking.c | 1131.7 | FA7C |
| 0 | 34 | edgeLoopY_S | deblocking.c | 1132.14 | FAA4 |
| 102 | 0 | edgeLoopC_N | deblocking.c | 1184.3 | FB50 |
| 1176 | 0 | edgeLoopC_N | deblocking.c | 1197.8 | FBB4 |
| 10510 | 0 | edgeLoopC_N | deblocking.c | 1198.8 | FBBC |
| 10498 | 0 | edgeLoopC_N | deblocking.c | 1199.8 | FBC0 |

This shows that for example in the function *getNextVidAUH264()* at source code lines 347 to 349 (Listing 10), store accesses occur with addresses adjacent to the accesses before. The program code at this location shows that the four accesses can be easily combined to one word accesses:

```
345 :  /* write start code to data buffer */
346 :  *p8_data++ = 0;
```

```
347 :  *p8_data++ = 0;
348 :  *p8_data++ = 0;
349 :  *p8_data++ = 1;
```

*Listing 10: Code example with successive byte accesses*

### 4.3.2 Using Caches and Non-Cacheable Areas

Especially for systems with slow memory, caches are mandatory for achieving a reasonable performance. The spatial and temporal locality of memory accesses found in most applications can be used efficiently with caches. On the other hand, if data areas are accessed randomly, for example in look-up tables, these accesses lead to cache trashing, i.e. a one-time accessed data value replaces a frequently accessed data value, which subsequently needs to be reloaded. The cache control unit usually allows the definition of non-cacheable areas in order to prevent data from these areas from being stored in the cache.

Obviously, achieving information about the access patterns to specific memory areas requires a dynamic analysis of the memory accesses. In such an analysis, the ratio between accesses to a data area and the resulting cache misses needs to be evaluated.

$$cacheMissRatio = \frac{cacheMisses}{MemoryAccesses} \tag{5}$$

### 4.3.3 Page Miss Reduction in DRAMs

The external memory of an embedded system is often dynamic RAM (DRAM), which is organized as pages, see Section 3.1.2. If a specific page is active, memory accesses to this page (page hits) are fast, whereas accesses to other pages (page misses) require several initialization steps, which results in wait states. Therefore data should be arranged such that the number of page misses can be reduced. A typical case where page misses occur is when two data areas are accessed alternatingly, for example when calculating the sum of two arrays. Each data access leads to a page miss or, if a cache is used, each line fill leads to a page miss. If possible, both data areas should be placed in one page.

As data placement is performed during compile time (or even runtime for heap variables), identifying page misses requires dynamic profiling of the running code. The MEMTRACE profiler provides page miss results for each function and for global or user defined data area and thus allows an identification of the code segments and data areas that should be rearranged.

### 4.3.4 Speedup Estimation before Implementation

Optimizing the software, e.g. by using SIMD instructions, assembler inlines or general re-coding, can be very time-consuming and prone to error. Therefore it is helpful to estimate the speedup that can be achieved by re-coding a specific function, and its influence on the overall performance, before performing the re-coding. MEMTRACE allows the specification of a speedup factor for each function in the application. Thus the influence of optimizing a specific function on the overall performance can be estimated.

### 4.3.5 Data Access Visualization

Beside the statistical representation of the profiling data, a visualization of the memory accesses gives the designer a good overview of the access pattern of the software over time. The information can be used to gain a better understanding of the memory accesses of the software and allows the finding of patterns within the accesses in order to optimize the data transfers, e.g. by prefetching subsequently used data to the cache or fast memory. Additionally, unnecessary data accesses are much easier to track. Figure 21 shows a set of screenshots of such access pattern images. Each image shows a snapshot of the accesses to an area in the memory (176x16 bytes) between a timing interval, e.g. between each call of a specific function. The number of accesses are then normalized to a 256-step wide gray-scale range and written continuously to a datastream. This datastream can then be interpreted as a luminance video stream and rendered by a video player.

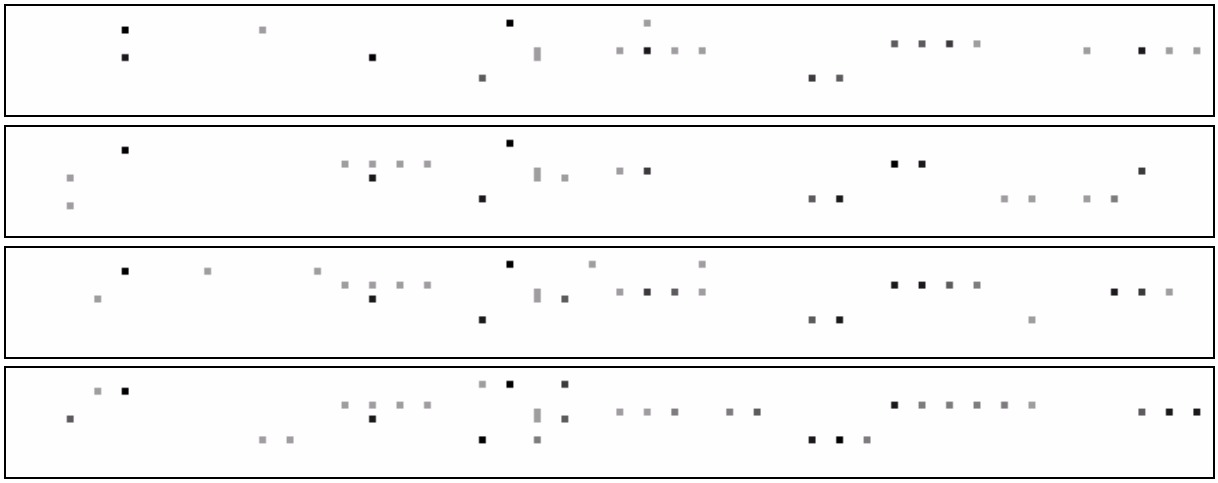


*Figure 21: Visualization of access pattern to user-defined memory area over time*

### 4.3.6 Efficient Register Usage

As memory accesses are very time-consuming, frequently accessed variables should be kept in registers if possible, as described in [12]. Register allocation for C source code is automatically performed by the compiler. However, the compiler may allocate registers inefficiently if global variables, pointers or pointer chains are used, as the accessed variables may be modified between multiple accesses. If the programmer knows that a variable is not modified, the compiler can be directed to use a register to store the variable by various methods, e.g. by defining it as a “register” type or by working with a local copy of a global variable.

An indicator of inefficient register usage is if a function accesses the same memory address multiple times. To localize such functions, the memory accesses in each function need to be analyzed. For each memory address accessed from the function, the number of accesses is counted. If a large number of accesses occurs to the same address, the above-mentioned methods might be applicable. Table 7 shows an excerpt of such results as provided by the profiling tool.

*Table 7: Access statistics for each function*

| Evaluated functions | Calls | Accessed addresses | Address #1 | Source file | Line in source file | Accesses * calls | Max. no. of accesses | Address #2 |
|---|---|---|---|---|---|---|---|---|
| **Itrans** | 144 | 68 | F2D74 | block.c | 108.5 | 4752 | 33 | … |
| **ReadCoeffBlockCAVLC** | 202 | 25 | stack | cavlc.c | 840.4 | 2828 | 14 | … |
| **PredictNnz** | 176 | 4 | F25EB | cavlc.c | 609.3 | 9152 | 52 | … |
| **ReadLevelVLCN** | 65 | 17 | stack | cavlc.c | 557.5 | 520 | 8 | … |
| **ReadTotalZeros** | 90 | 6 | stack | cavlc.c | 232.1 | 720 | 8 | … |
| **DecodeH264** | 1 | 16 | stack | decodeH264.c | 388.1 | 11 | 11 | … |
| **GetVLCSymbol** | 430 | 7 | stack | egvlc.c | 151.1 | 4730 | 11 | … |
| **GetVLCSymbol_Slow** | 20 | 4 | stack | egvlc.c | 121.1 | 40 | 2 | … |

For each function, the table gives the number of calls and the total number of accessed addresses. For the ten most accessed addresses, more details are provided. Table 7 shows only the results for the first address as an example. The results include the actual address (or location name), the number of accesses per function and the number of accesses multiplied by the number of calls. For this last, the call of the function with the highest number of accesses is used. This number can be used to compare the overall influence of accessing this address, as compared to accesses in the other functions. If source code information is available, the profiler also provides one source code line location, where an access to the memory address occurred. This helps in identifying the actual source code variable which corresponds to the address, and is a candidate for the manual register allocation.

For example, the function “itrans” in the first line of the table is called 144 times. The function accesses up to 68 different memory addresses during each call. The most frequently called memory location is at address 0xF2D74. The call (or at least one of the calls) to this address takes place at line 108 in the source code file “block.c”. The maximum number of accesses to this address within one call of the function is 33 times. The overall influence of the accesses to this address is assessed by multiplying the maximum number of accesses by the number of calls, i.e. 33 times 144, which results in 4752. The higher this number is, the larger the influence on the overall performance when the data value of this address is stored in a register. This fact becomes clearer when comparing the two functions “DecodeH264” and “getVLCSymbol”. Both functions access a specific address on the stack 11 times. However as the “getVLCSymbol” function is called 430 times, optimizing the register allocation there yields a higher speedup than optimizing “DecodeH264”, which is only called once.

## 4.4 Profiling-Based Hardware Optimization

Beside the software optimizations presented in the previous section, profiling can also be applied to adjust the processor and memory architecture of embedded systems. The following sections show how the instruction set and the address generation modes of a processor can be adjusted to the needs of the application. In Section 4.4.3 a method is described for configuring and using fast on-chip memory efficiently.

### 4.4.1 Instruction Set

As their name states, RISC processors come with a reduced instruction set as compared to CISC processors. However, some of the current RISC instruction sets provide more than 100 instructions. If the instruction set of the processor is customizable, such as with the CoWare Processor Designer, Tensilica or ARC, it can be helpful for the processor designer to obtain information about the actual usage of the instruction set. A statistical analysis can be performed by parsing the compiler-generated assembly code. However, this static analysis neglects the real instruction usage during program execution, since not every assembly code line is executed equally often. As many instructions can be replaced by a series of other instructions, it can be helpful to see how often a specific instruction is really used. This is important as the replacement with other instructions often comes with an overhead, and therefore the influence of the overhead can be estimated by this dynamic profiling.

Table 8 shows the instruction profiling results as provided by the profiler for the execution of an H.264/AVC decoder. The source code, which includes more than 20,000 lines of code, is translated to a usage of only 21 assembly instructions. Thus, a processor design with only these instructions would be sufficient to execute the code. Furthermore, it can be seen that four instructions (LDR, ADD, STR, MOV) are responsible for almost 75 % of the decoded instructions. So, the processor architecture, including the instruction set and decoder, pipeline and memory interface should be designed such that these instructions require a very low latency. The fact that these instructions are mainly data movement related shows how data-intensive this application example is.

### 4.4.2 Address Modes

Besides the actual instructions, the instruction set of a processor is also defined by the address modes which are implemented. The addresses are either calculated in a separate address generation unit or within the general ALU including the shifter unit. Depending on the processor, a more or less wide range of address modes is available. Taking the ARM architecture as an example, the following modes are supported. The simplest is a zero offset address, where the address is taken from a register without any offset. This address can be modified by an immediate value, which is hard-coded in the instruction. The register values can be further processed by a shift operation within the same instruction.

Supporting all these address modes has two major impacts on the processor architecture. On one hand, the coding of the address modes in the instruction set requires a portion of the instruction bit-width for encoding the mode (3 bits), the offset register (4 bits), shift information (7 bits) and the immediate value (12 bits). On the other, the hardware support required for calculating the addresses leads to an overhead in die area and power consumption. This is especially true if a separate address generation unit is used. If a processor is targeted to a specific application, the architecture should be adapted to the application's needs. A profiling of the applied address modes can be used to build an optimized and reduced instruction set and address generation unit.

*Table 8: Instruction profiling results*

| Instruction | Executed | Percent | Accumulated |
|---|---|---|---|
| **SWI** | 734 | 0 | 100 |
| **ADC** | 19859 | 0.01 | 100 |
| **BIC** | 28336 | 0.02 | 99.99 |
| **CMN** | 35269 | 0.02 | 99.97 |
| **MVN** | 422525 | 0.29 | 99.95 |
| **TST** | 514008 | 0.35 | 99.66 |
| **RSB** | 872541 | 0.59 | 99.31 |
| **ORR** | 938291 | 0.64 | 98.72 |
| **LDM** | 1655893 | 1.13 | 98.08 |
| **MUL** | 1698881 | 1.16 | 96.95 |
| **STM** | 1709378 | 1.16 | 95.79 |
| **AND** | 2742492 | 1.87 | 94.63 |
| **MLA** | 2945366 | 2.01 | 92.76 |
| **SMU** | 3494211 | 2.38 | 90.75 |
| **SUB** | 6097076 | 4.15 | 88.37 |
| **B** | 6974355 | 4.75 | 84.22 |
| **CMP** | 7171197 | 4.88 | 79.47 |
| **MOV** | 15562089 | 10.59 | 74.59 |
| **STR** | 16698445 | 11.37 | 64 |
| **ADD** | 35844633 | 24.4 | 52.63 |
| **LDR** | 41458397 | 28.23 | 28.23 |
| **Sum** | 146883976 | 100 % | 0 |

Table 9 shows the example results of a profiling run for a gesture recognition system, see Section 6.2. For each of the load and store operations one of the address types is used, with either no offset at all, a program counter relative offset or a pre- or post-indexed offset. These offsets can be either an immediate value or taken from a register value. Furthermore, the register value can be shifted by a given value and a specific shift operation. As can be seen, here most of the memory accesses are to pre-indexed addresses with an immediate offset value. Register offsets are used for less than 12 % of the memory accesses and shift operations for only 2 %. For optimization purposes, abandoning these address modes could be an option.

*Table 9: Address mode profiling results*

| **Details on load and store operations** | |
|---|---|
| **Loads** | 2751552 |
| **Stores** | 740142 |
| **Address type** | |
| **Zero-offset** | 663579 |
| **Program counter-relative** | 58324 |
| **Pre-indexed** | 2188188 |
| **Post-indexed** | 581603 |
| **Detail on all indexed modes** | |
| **Immediate offset** | 2352885 |
| **Register offset** | 416906 |
| **Detail on register offset (optional shift operation)** | |
| **Shift-offset ASR** | 0 |
| **Shift-offset LSL** | 75361 |
| **Shift-offset LSR** | 746 |
| **Shift-offset ROR** | 0 |
| **Shift-offset RRX** | 0 |

### 4.4.3 Data Partitioning between Fast and Slow Memory

Many embedded system architectures provide a fast but small internal memory (SRAM) as an addition to the much slower external memory (DRAM). This internal memory can be used to store frequently used data for fast access. As SRAM is very costly in die area and power consumption, the internal memory is usually small. Therefore in order to use it efficiently, the frequently accessed memory areas need to be identified, these being valuable candidates for internal storage.

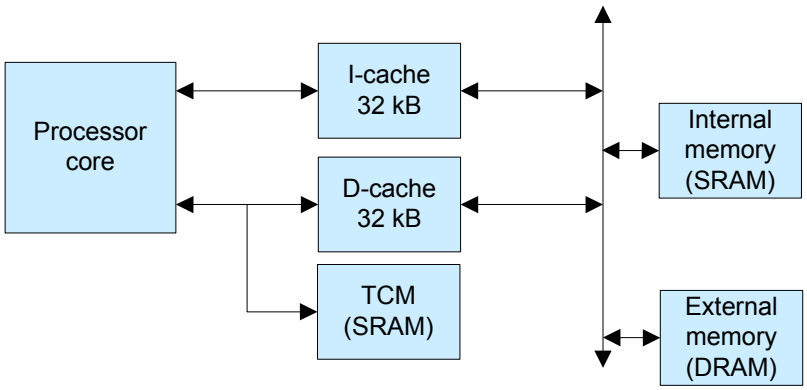



*Figure 22: Embedded system with caches and fast internal and slow external memory*

If caches are available in a system, as depicted in Figure 22, the situation changes slightly, as not every load/store access is passed to the slow external memory. Instead, this occurs only if the required data/instructions are not available in their caches, i.e. cache misses occur. The caches then load new cache lines (e.g. eight words) from the external memory (DRAM). These loads lead to a halt of the processor and thus increase the execution time. The time required for a cache load depends on the speed (wait states) of the external memory. Thus the overall number of cache misses must be reduced in order to speed up the application.

This can be accomplished by using one of the fast internal memory devices. In cases where a TCM is used, the number of cache accesses (and misses) is reduced directly. In the case of other internal memory, the address range of this memory should be marked as "non-cacheable", in order to bypass the cache during accesses to this memory. The choice of data/instructions to be stored in the TCM is quantified by the number of cache misses which occur when accessing these data/instructions. Therefore an analysis of cache misses per data segment (e.g. variable) or instruction segment (e.g. function) is required. The resulting cost ratio, given in Equation 6, expresses the ratio between the cache misses that occur when accessing a data segment and the size of the segment.

$$cost\ ratio = \frac{cache\ misses}{data\ size} \tag{6}$$

The areas with the highest cost value should be stored in fast memory in order to reduce the overall number of misses.

The example shown in Table 10 is used to describe the mapping method. The table shows an example list of variables with the results for size and cache misses, sorted by the cache misses per byte. The accumulated size and number of cache misses is also given, with the results of data structures accumulating from the top to the bottom of the list. With a given TCM size, for example 4096 bytes, all data structures from "incVlc" down to "predictIntra4_table" could fit into the TCM. However, this leads to using only 2568 bytes of the 4 kB available. A more efficient method would be to leave "predictIntra4_table" out and thus have space for "expgolombtab", which would increase the number of saved cache misses from 96,849 to 142,807.

The process of finding an optimal partitioning of the data segments to the fast and slow memory, while reducing the overall cache misses, can be described with the knapsack problem [65]. This can be described as follows: having a container with a capacity $c$ and a number $n$ of objects that can be either taken or not (x=0 or x=1), with each object having a weight $w$ and a profit $p$, the objective is to

$$\text{maximize } \sum_{j=1}^{n} p_j \cdot x_j$$

$$\text{subject to } \sum_{j=1}^{n} w_j \cdot x_j < c$$

Here the data size is equivalent to the weight and the cache misses are the profit. Although solving this problem can be complex, especially when dealing with a large number of objects, efficient algorithms exist to reduce the complexity.

*Table 10: Decoder data areas sorted according to their number of cache misses per byte*

| Data structure | Size | Load, store accesses | Cache misses | Accum. size | Accum. cache misses | Cache misses per byte |
|---|---|---|---|---|---|---|
| **IncVlc** | 28 | 116142 | 1187 | 28 | 1187 | 42.39 |
| **Getqpel8_table** | 64 | 32608 | 2638 | 92 | 3825 | 41.22 |
| **Run0_short** | 128 | 52206 | 5195 | 220 | 9020 | 40.59 |
| **NumCoeffTrailOnes0** | 1024 | 242918 | 39531 | 1244 | 48551 | 38.6 |
| **Run1_short** | 128 | 34840 | 4870 | 1372 | 53421 | 38.05 |
| **Bs** | 4 | 3213784 | 150 | 1376 | 53571 | 37.50 |
| **ALPHA_TABLE** | 52 | 402619 | 1933 | 1428 | 55504 | 37.17 |
| **TotalZeros0_short** | 1024 | 102493 | 37300 | 2452 | 92804 | 36.43 |
| **TotalZeros** | 60 | 216663 | 2100 | 2512 | 94904 | 35.00 |
| **PredictIntra4_table** | 56 | 146961 | 1945 | 2568 | 96849 | 34.73 |
| **Expgolombtab** | 1540 | 656242 | 47903 | 4108 | 144752 | 31.11 |
| **…** | … | … | … | … | … | … |

The resulting partitioning of the data segments can be used for automatic placement if a feedback path from the profiler to the compiler or linker, respectively, can be established. For example, by means of a linker script or scatter loading files, the placement of each object file can be achieved. To place each variable of the code individually, a compiler directive (pragma) can be inserted in the code. This forces the creation of special sections for each variable, which can then be accessed for placement within the scatter file.

The method described above leads to a static placement of the data segments. However, the profiling can also be used for dynamic placement to the fast local memory, which corresponds to the actual purpose of TCMs. If the profiler is instructed to produce intermediate results, these results can be used to analyze the changes in data usage and cache miss behavior over time. As described in Section 4.1, the “split function” behavior can be used to produce profiling snapshots at each call of a specific or even of multiple functions.

The following is an example of dynamic placement. If an array or look-up table is used heavily only within one function but not required during other functions, it can be loaded to the TCM dynamically only for the time it is used, leaving the memory space for other variables at other times. A DMA controller should be used in order to perform the relocation of the array efficiently and the programming of the controller can be based on the profiling results.

## 4.5 Power Model of an Embedded Processor

Besides providing performance and memory analysis, the profiling method has been extended with power estimation. For this purpose a measurement-based model has been created. Compared to analytical models, measurement-based models have the disadvantage of being restricted to a specific processor. On the other hand, measurement leads to real power numbers and thus is proven to be valid. Processor cores are not usually manufactured as stand-alone devices, but are incorporated onto a device along with additional components such as on-chip

memory, control units and input/output modules. The influence of these components on power consumption needs to be considered too.

Within this study, the Altera Excalibur [1] platform has been used for measuring the power consumption of a sample embedded RISC processor core. It is an SoC based on an ARM922T processor with separate instruction and data caches, each 8 kB, see Figure 23. Besides the processor, single and dual port on-chip SRAMs are available and memory controllers for accessing external SDRAM and a field-programmable gate array (FPGA) are contained on the chip.

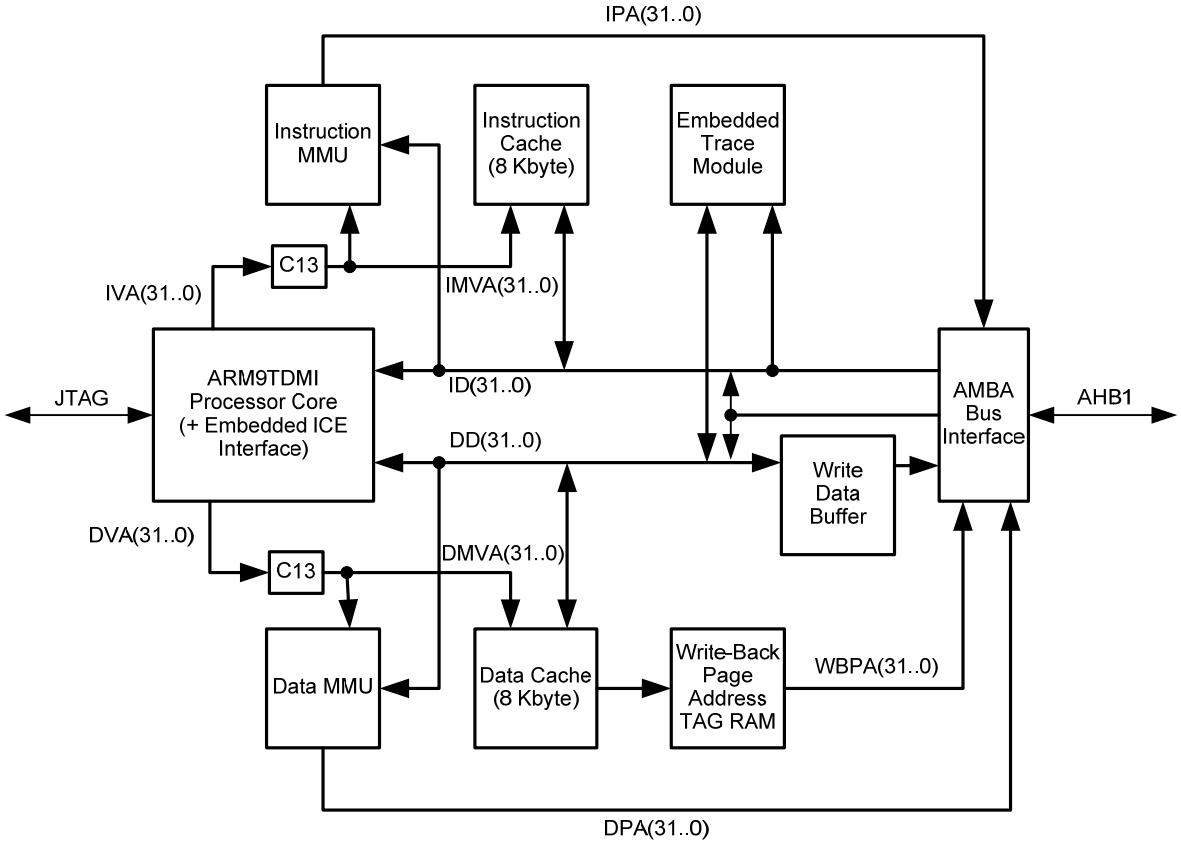



*Figure 23: Block diagram of the Excalibur processor unit including caches and MMUs [1]*

The Excalibur device is built in a 0.18 µm TSMC process with seven metal layers. The PLD array is similar to the arrays used in APEX 20KE devices. The Excalibur series features three devices, which differ in their size of on-chip SRAM and gate array. For the power evaluation of the processor, the smallest device, called EPXA1, has been chosen, in order to keep the influence of the other on-chip components low.

### 4.5.1 CMOS Power Consumption

Most embedded systems are built of semiconductor devices, which use the complementary MOS (CMOS) technology. The power consumption of CMOS circuits is a sum of static and dynamic components. The dynamic component describes the switching activity of the circuit, which is mainly due to the charging and discharging of load capacitances. These capacitances include the interconnecting wires and the internal capacitances of the transistors, therefore the power consumption increases with the wire length and the number of connected gates. The dynamic power consumption is defined as:

$$P_{dynamic} = C_k \cdot f_k \cdot V_{DD}^2 \tag{7}$$

where $C_k$ is the load capacitance, $f_k$ the clock frequency of the circuit and $V_{DD}$ the supply voltage.

When the CMOS technology was introduced, one of its major advantages was the very low static power consumption, which is due to the fact that either the pull-up (PMOS network) or pull-down (NMOS network) of a gate is turned off when the circuit is in a stable state. The continuous shrinking of the feature size of the transistors and the accompanying reduction of the threshold voltage leads to an increased leakage current, the subthreshold leakage, because the gate's transistors are not fully turned off in this state. Therefore in current CMOS technologies, the static power consumption can no longer be neglected, and in the future it will become comparable to the dynamic power consumption [103]. Therefore the power model created within this work reflects both, dynamic and static power consumption.

### 4.5.2 Power Measurement Methods

The electrical power consumed by a device is defined as:

$$P(t) = V(t) \cdot I(t) \tag{8}$$

where $V$ is the supply voltage and $I$ is the electric current flowing into the device. For semiconductor devices, the supply voltage is provided in DC mode and can be considered constant. This leads to

$$P(t) = V \cdot I(t) \tag{9}$$

Thus the power consumption of the device can be determined by measuring the current flowing through the device. The current changes depending on the chip activity and the energy consumed within a time $\Delta t$ is expressed as:

$$E = \Delta t \cdot \int P(t) = \Delta t \cdot \int V \cdot I(t) = \Delta t \cdot V \cdot \int I(t) \tag{10}$$

The electrical current can only be measured indirectly by one of the following three methods, which are described in more detail below:

- voltage drop over a shunt resistor
- magnetic field produced by current flowing through a conductor
- voltage drop due to discharging of switching capacitors [28]

The shunt resistor setup shown in Figure 24 is the most common current measurement method. It has a small resistance, usually in the range of a few milliohms to 1 ohm. The resistor is installed in series with the system to take measurements on the power supply line. It can be installed either on the supply voltage side (high-side) or on the ground side (low-side) of the power supply. The low-side arrangement has the advantage that no common mode voltage exists, however it might be difficult to measure all ground paths, for example ground path might also appear through the measurement equipment or other connected devices. The current mode voltage of high-side measurements can be eliminated by means of a differential measurement setup. This can be either a differential probe or a differential amplifier, which can also be used to amplify the signal.

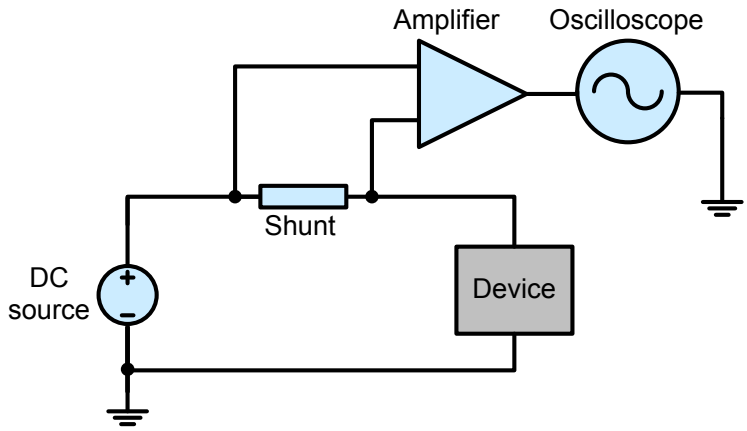



*Figure 24: Measurement setup with high-side shunt resistor and amplifier*

The problem with the shunt setup is that it is an intrusive method. The dynamic voltage drop over the shunt (due to the changing supply current) leads to a changing supply voltage to the actual device. If the supply voltage is too low, this can lead to malfunction of the device. Usually CMOS devices have a specific supply voltage range which is allowed, and this is the maximum range for the actual measurement. As this range is usually about a few millivolt, e.g. 10 - 100 mV, the actual measurement device must be fairly accurate. Therefore it is helpful to amplify the signal, although this in turn leads to a reduced frequency range due to the frequency and gain relationship of the operation amplifier. This relationship is given in the gain bandwidth product. A trade-off has to be found between bandwidth and amplification, which corresponds to a trade-off between temporal and signal range accuracy. Simple operational amplifiers can be used as differential amplifiers, if they are setup as shown in Figure 25.

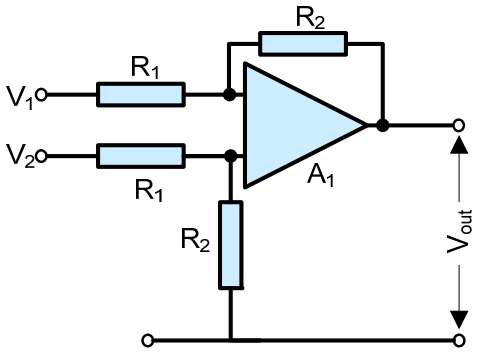



*Figure 25: Schematic of a differential amplifier [20]*

The drawback of this setting is that it has only medium input impedance. As the amplifier needs to be connected in parallel to the shunt resistors, this can lead to errors in the measurement results. Instrumentation amplifiers overcome this problem by extending the circuit with two input operational amplifiers, see Figure 26. This leads to a high input impedance, in the range of gigaohm, and a high common mode rejection.

Instrumentation amplifiers are available as integrated circuits, which offer a higher accuracy than discrete circuits, for example the Analog Devices AD623 [4] used in this work. The AD623 integrates the entire circuit given in Figure 26 except for the resistor controlling the gain, which can be externally connected to the device. The gain bandwidth product is 800 kHz.

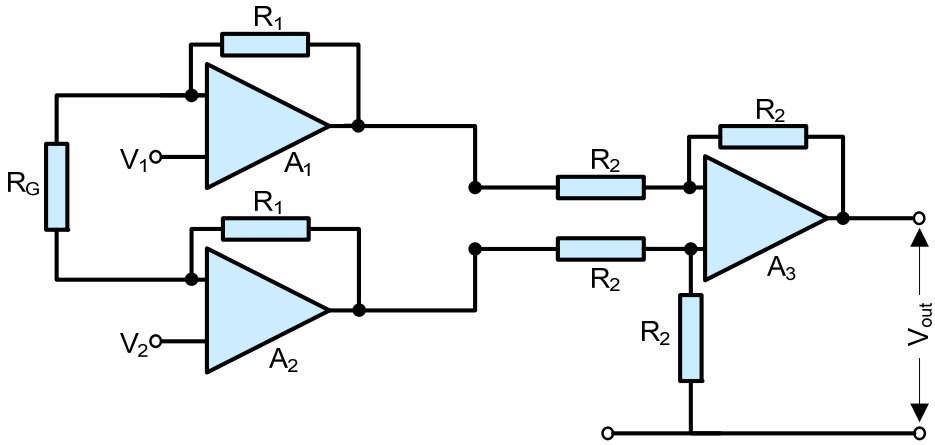



*Figure 26: Schematic of an instrumentation amplifier [20]*

Especially for the measurement of electric current, so-called current-sense amplifiers are available which provide a higher gain-bandwidth product. For example, the Maxim MAX4376TASA offers 40 MHz and has a bandwidth of 2 MHz at a gain of 20. The operation mode is different from the instrumentation amplifier, see Figure 27.

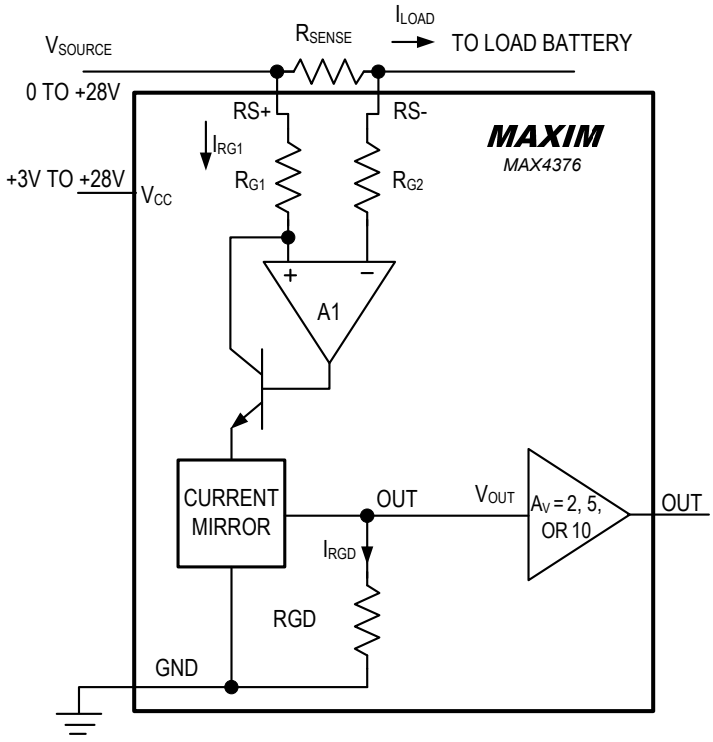



*Figure 27: Block diagram of the current-sense amplifier MAX 4376 [67]*

The current to be measured, $I_{load}$, leads to a voltage drop over $R_{Sense}$. The open-loop setup of the operational amplifier leads to the same voltage drop over $R_{G1}$ and results in current $I_{RG1}$.

$$I_{RG1} = \frac{1}{R_{G1}} \cdot R_{Sense} \cdot I_{Load} \tag{11}$$

This current is "copied" and amplified by a factor $\beta$ with a current mirror, which leads to the current $I_{RGD}$. The voltage drop over the resistor $R_{GD}$ is then amplified by factor $A_v$ of 2. The output voltage is then:

$$V_{Out} = \beta \cdot \frac{R_{GD}}{R_{G1}} \cdot R_{Sense} \cdot I_{Load} \cdot A_V \tag{12}$$

The combined gain factor is given in Equation 13 and is adjusted to a value of 20 for the MAX4376TASA device.

$$gain = \beta \cdot \frac{R_{GD}}{R_{G1}} \cdot A_V \tag{13}$$

The instrumentation amplifiers mentioned above and the current-sense amplifier are used in a high-side setup for the current measurement. The measurement setup is described in more detail in Section 5.7.

### 4.5.3 Instruction Sequences for Power Evaluation

The power model of the processor should be triggered by information which is delivered by the profiler or by the ISS. This includes, for example, the executed instruction, register usage, pipeline stalls, data values and addresses and cache misses. In order to create the power model, the influence of these parameters on the power consumption needs to be extracted. Based on existing power models, as described in Section 3.2.1, the following typical influences on the dynamic power consumption have been identified:

- intra-instruction energy
- inter-instruction energy
- data and instruction cache
- data path
- register address decoder
- idle cycle

The intra-instruction energy corresponds to the intrinsic energy for each instruction without any influence of other instructions. This energy corresponds to the activity in the functional units of the processor, when they are utilized by an instruction. This is covered by running a program which consists of a sequence of only this instruction. Interdependencies between instructions are reflected by the inter-instruction energy. This energy is due to the activation and deactivation of specific functional units. A test program should cover each combination of instructions by executing them alternatingly. The energy as compared to the single instruction sequence is the inter-instruction offset.

Besides the actual instructions, the influence of the processed data streaming through the data path should also be considered. For abstracting the data dependencies two different models are examined: the Hamming distance and the number of ones. The Hamming distance describes the difference between two successive data values, expressed as the number of bit positions which have changed from one to the next. This behavior corresponds very well to the charging and discharging process of the data bus wires and therefore should correlate with the energy consumption. Depending on the charging behavior of the wires, the number of ones within a data word can be considered as an additional criterion for an energy model. In order to profile the influence of the data path, the data read from the source registers and the final data written to the destination register need to be profiled.

These aspects apply similarly for the binary representation of the instruction and the addresses for accessing data, instruction and registers.

Since the register file tends to have a high influence on the data path energy, the influence of the address decoder of the register file should be considered along with the actual data flow. This is done by using a sequence of instructions with changing source and destination registers.

Instead of examining each cache separately, the cache energy estimation can be combined with that of the processor by profiling the cache access behavior. This simplification is feasible for a simple processor, where every activity of the cache is initiated by the processor. The data cache is activated if a load or store operation is executed and the instruction cache is activated during an instruction fetch. For the data cache, the data and addresses transferred during each load or store operation need to be considered. For the instruction cache, an instruction sequence needs to be executed, which leads to a specified Hamming distance between the instructions and a defined number of ones. However, the instructions within the sequence should activate similar functional units in order to separate this influence from the cache model. Furthermore, the cache hit and miss behavior needs to be taken into account. For more complex architectures, the cache should be modeled separately, for example by using the CACTI cache power model, as mentioned in Section 2.3.4.3.

Finally, the different idle modes need to be examined. Processors can be idle either during a NOP operation or when a pipeline stall occurs due to a cache miss. Additionally, many processors support special low power modes such as standby, where most of the functional units are turned off by clock gating.

Since the energy model needs to be incorporated into the profiler, it should rely on the data provided by the profiler or by the cycle-accurate ISS. Considering the aspects described above, the following profiling data are required:

- executed instruction
- accessed, processed and written back data from the register
- instruction word and address read from the instruction cache
- data word and its address accessed during load and store operations
- cache behavior, including cache misses and hits as well as write back stalls
- timing behavior, i.e. the delay and throughput of instructions, pipeline stalls and delay time for cache hits and misses
- accessed registers and register addresses

A set of assembly code sequences needs to be developed which reflect the different energy contribution. These code sequences can then be run on the processor to evaluate the dynamic behavior of the processor. Section 5.8.2 describes this process in more detail.

### 4.5.4 Power Model of an SoC

The results show that the instructions can be grouped in classes of instructions with similar power consumption. For example, all simple arithmetical instructions, such as ADD, SUB and CMP have the same power consumption. The measurements performed result in the following model creation. For the chosen SoC design, the overall energy consumption on core supply voltage can be expressed as:

$$P_{Core} = P_{CPU} + P_{Caches} + P_{FPGA} + P_{SDRAM-Controller} + P_{On-Chip-SRAM} \quad (14)$$

The FPGA array is disabled and can be assumed to have constant power consumption. The same applies for the on-chip memory devices, as they are not used during the measurement. The SDRAM controller is active if a cache miss occurs. It is also active for some maintenance tasks, such as refresh, although these are assumed to have a negligible effect. The power consumption of the CPU and of the caches is separated into a constant part and a dynamic part, depending on the activity. Furthermore, the cache activity is separated into cache miss and cache hit. Thus the power consumption of the SDRAM controller is directly dependent on the cache misses.

$$P_{Core} = P_{CPU,dynamic} + P_{I-Cache\ hit} + P_{D-Cache\ hit} + P_{I-Cache\ miss} + P_{D-Cache\ miss} + P_{const} \tag{15}$$

and

$$P_{const} = P_{On-Chip-SRAM} + P_{FPGA} + P_{CPU,Idle} + P_{Caches,Idle} \tag{16}$$

The cache miss power consumption includes the cache and the SDRAM controller activity, as it is directly dependent on the cache misses. The dynamic part of the power consumed by the CPU is dependent on the executed instruction.

Taking the current measurement into account for the different dynamic influences, the following energy consumption models are established:

$$E_{CPU,dynamic} = \sum_{ExecutedInstr} V_{DD} * I_{Instr} * t_{Instr} \tag{17}$$

Taking into consideration the data dependencies, the CPU portion can be calculated with

$$E_{CPU,dynamic} = \sum_{ExecutedInstr} V_{DD} * t_{Instr} * (I_{Base} + I_{Hamm} * Hamm + I_{Reg1s} * Reg1s) \tag{18}$$

The energy consumption of the cache and the SDRAM-controller is modeled depending on number of cache hits and misses as given in Equation 19 and 20.

$$\begin{aligned} E_{I-Cache} = {} & V_{DD} * I_{Cache-Miss} * t_{Cache-Miss} * N_{Cache-Miss} \\ & + V_{DD} * I_{Cache-Hit} * t_{Cache-Hit} * N_{Cache-Hit} \end{aligned} \tag{19}$$

$$\begin{aligned} E_{D-Cache} = {} & V_{DD} \cdot I_{D-Cache-Miss} \cdot t_{D-Cache-Miss} \cdot N_{D-Cache-Miss} \\ & + V_{DD} \cdot I_{D-Cache-Hit} \cdot t_{D-Cache-Hit} \cdot N_{D-Cache-Hit} \end{aligned} \tag{20}$$

The power model has been incorporated into the profiler. This allows dynamic energy estimation for an application. Table 11 shows sample results provided by the profiler after analyzing an application run on the ISS. A list of all instruction types executed is given along with their contribution to the energy consumption. For example, the add (“ADD”) instruction in the second row was decoded 14,641,240 times. 13,988,435 of these add instructions were executed, while the remaining 652,805 were skipped. The skipping of instructions is due to the conditional execution features of the ARM processors, i.e. an instruction is only executed if the conditional flags are met. The energy required for processing all ADD instructions (executed and skipped) is estimated to be 42.035 mJ. This corresponds to an average current value of 319 mA on the core voltage supply line.

*Table 11: Energy estimation based on the instruction and data transfer profiling results*

| Instruction | Decoded | Executed | Skipped | Energy (mJ) | Current (mA) |
|---|---|---|---|---|---|
| **ADC** | 38343 | 38343 | 0 | 0.110083 | 319 |
| **ADD** | 14641240 | 13988435 | 652805 | 42.035 | 319 |
| **AND** | 2393132 | 2271771 | 121361 | 6.87068 | 319 |
| **B** | 7350494 | 5079545 | 2270949 | 52.3186 | 332 |
| **BIC** | 14077 | 14032 | 45 | 0.0404151 | 319 |
| **BL** | 1455186 | 1453595 | 1591 | 13.0348 | 332 |
| **BX** | 461190 | 439665 | 21525 | 4.00547 | 332 |
| **CMN** | 8532 | 8532 | 0 | 0.0244954 | 319 |
| **CMP** | 7728864 | 7644032 | 84832 | 22.1896 | 319 |
| **EOR** | 4 | 0 | 4 | 1.15E-05 | 319 |
| **LDM** | 1412459 | 1297982 | 114477 | 4.51281 | 355 |
| **LDR** | 18382820 | 17930740 | 452080 | 59.2294 | 358 |
| **MLA** | 622894 | 622830 | 64 | 10.5945 | 315 |
| **MOV** | 11324464 | 10025492 | 1298972 | 32.5125 | 319 |
| **MUL** | 324356 | 324356 | 0 | 5.5173 | 315 |
| **MVN** | 481301 | 111801 | 369500 | 1.38182 | 319 |
| **NOP** | 430 | 430 | 0 | 0.00123453 | 319 |
| **ORR** | 1306312 | 987059 | 319253 | 3.75042 | 319 |
| **RSB** | 2596264 | 1670553 | 925711 | 7.45387 | 319 |
| **STM** | 1245727 | 1240185 | 5542 | 3.87919 | 346 |
| **STR** | 9196038 | 9001769 | 194269 | 59.1247 | 361 |
| **SUB** | 7532152 | 7418239 | 113913 | 21.6248 | 319 |
| **SWI** | 63 | 63 | 0 | 1.81E-04 | 319 |
| **TST** | 165372 | 165358 | 14 | 0.474783 | 319 |
| **Sum** | **88681714** | **81734807** | **6946907** | **350.68666** | |

| Others | Value | Per instruction/ per mem. access | Instructions/ mem. access | Energy (mJ) | Current (mA) |
|---|---|---|---|---|---|
| **Idle cycles** | 31870309 | | | 75.7239 | 264 |
| **HammingI** | 891757368 | 10.056/instruction | 88681714 | 3.00968 | |
| **HammingD** | 258309413 | 6.35/mem. access | 40675944 | 0.87179 | |
| **ALU_Trans** | 52426137 | 0.59/instruction | 88681714 | 5.662 | |
| **MEM_Trans** | 24859105 | 0.61/mem. access | 40675944 | 0.6712 | |
| **#1s_SrcReg** | 1894856992 | 21.367/instruction | 88681714 | 6.3951 | |
| **Overall sum** | | | | **443.02033** | **332** |

Additional influences, such as Hamming distance of cache accesses and inter-instruction dependencies, are shown at the bottom of Table 11, along with the overall estimation result. The result value (891,757,368) given in the "HammingI" row is the accumulated Hamming distance of all decoded instructions. This results in an average of 10.056 bits that have changed

in each of the 88,681,714 decoded instructions. These toggling bit lines on the instruction fetch and decoding units add 3.00968 mJ to the overall energy consumption. Similarly, the influence of the data memory accesses is provided in the row “HammingD”.

The inter-instruction influence on the energy consumption is given in rows “ALU_Trans” and “MEM_Trans”. The first corresponds to the transition from any instruction to an ALU instruction, whereas the latter describes the transition to a memory access instruction. These two types of inter-instruction transitions have been separated, as they lead to different energy consumptions. Transitions between two identical instructions are not counted, as they don’t add any extra energy consumption. For example the “ALU_Trans” row indicates that 52,426,137 of the 88,681,714 decoded instructions were transitions to ALU instructions, which corresponds to 59 % of the instruction. This inter-instruction effect adds 5.662 mJ to the consumed energy. The “#1s_SrcReg” row reflects the effect of the data flowing through the data path of the pipeline. The measurements have shown that the numbers of ones in the source register values are the indicators for this influence. Thus the data values of the accessed source registers are profiled and the numbers of ones in these values are accumulated. The “#1s_SrcReg” row shows that 1,894,856,992 ones were found in these values, which corresponds to an average value of 21.367 per instruction and adds 6.3951 mJ. The overall energy consumption of the entire application is estimated to be 443.02033 mJ, which incorporates the intra-instruction energy of each instruction and all the additional influences described above.

These results can be used to optimize the instruction set in cases where a customizable processor architecture is available. For example, infrequently used instructions might be replaced by a sequence of other instructions, in order to minimize the complexity of the instruction decoder. And instructions that dissipate a large portion of the overall energy should be considered as targets for optimization during the processor architecture development.

Besides the overall results for the entire application, MEMTRACE also delivers the energy estimation results on a function-accurate level. Table 12 shows an excerpt of the profiling results for a software implementation of the H.264/AVC decoder, described in Section 3.4.1. It lists the results for the C source code functions of the decoder, which are sorted in a descending order regarding energy consumption.

*Table 12: Profiling results of an H.264 video decoder (for one QVGA-sized frame), in descending order of energy consumption*

| Function | Calls | Cycles | Load/ store | Cache miss | Instruct-ions | Energy (uJ) |
|---|---|---|---|---|---|---|
| **EdgeLoopY_N** | 1364 | 896930 | 511808 | 2592 | 1493388 | 5601 |
| **H264_bzero** | 2003 | 437933 | 142216 | 0 | 582885 | 3178,7 |
| **MotionPredChroma** | 157 | 506157 | 238848 | 2639 | 418345 | 3032,2 |
| **Itrans** | 1707 | 351600 | 228305 | 3015 | 430164 | 2094,1 |
| **EdgeLoopC_N** | 565 | 287382 | 180949 | 1334 | 447759 | 1771,5 |
| **…** | … | … | … | … | … | … |

The most energy is consumed in the function “edgeLoopY_N”, which is part of the deblocking filter. Thus this is a good candidate for power consumption optimization. When the energy is compared with the cycles count, it can be seen that they correlate in most cases, thus the energy is often dependent on the execution time. This fact has also been observed by the

developers of JouleTrack, as described in Section 2.3.4.1. In some cases, for example when comparing the functions “h264_bzero” and “motionPredictionChroma”, the behavior is different. This is due to the fact that the latter function accesses the memory very randomly, which leads to the high amount of cache misses. Thus, a higher amount of the cycles required in “motionPredictionChroma” are idle wait cycles, which require less energy. Therefore, especially for data-intensive applications, energy optimization can not only be based on the clock cycles counts, but also on the influence of memory accesses.

Additionally to the overall energy consumption results, the dynamic behavior of the power consumption is of interest, for example to find peak values. Figure 28 depicts the estimation of the power consumption during the decoding of one video frame of an H.264/AVC video stream. It starts with the reading of the bitstream from hard disk to memory, followed by the actual decoding. The decoding is executed as a loop over all macroblocks. After the decoding, the deblocking takes place, which also loops over all macroblocks. As can be seen, during deblocking the power consumption rises significantly. The major functions of the deblocking are the “edgeLoop” functions found in Table 12. In these functions the ratio of cache misses (and with that also idle cycles) to the overall cycle count is significantly lower than in other functions of the decoder. Thus during the deblocking, most of the time the processor is busy and requires more power.

As a comparison, Figure 29 depicts the results during the measurement of the power consumption on the processor. The figure shows the screenshot of the oscilloscope during the decoding of an H.264 video stream, and the processing of approximately one video frame can be seen. A difference between estimated results and measured results is the bitstream reading at the beginning of each frame. This is due to the fact that in the hardware setup the bitstream already resides in memory before the decoding starts.

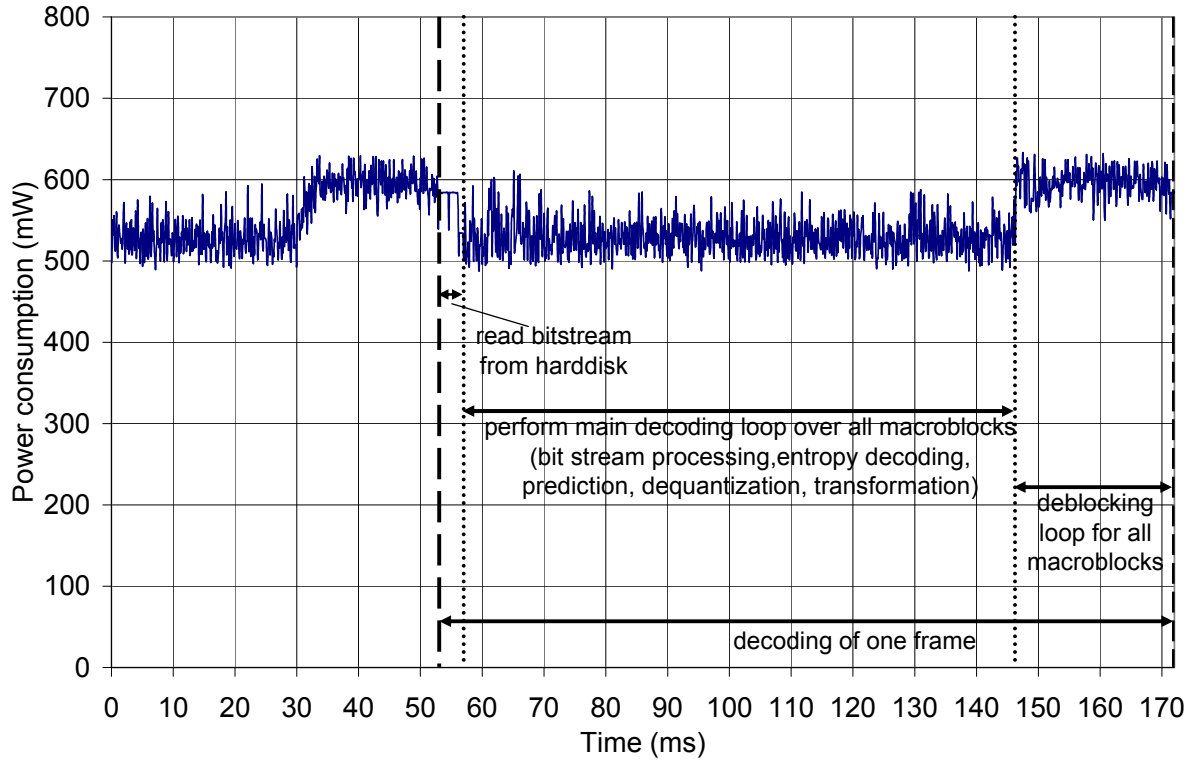



*Figure 28: Power consumption estimation results for the core supply voltage over time while decoding one video frame of an H.264 video stream*

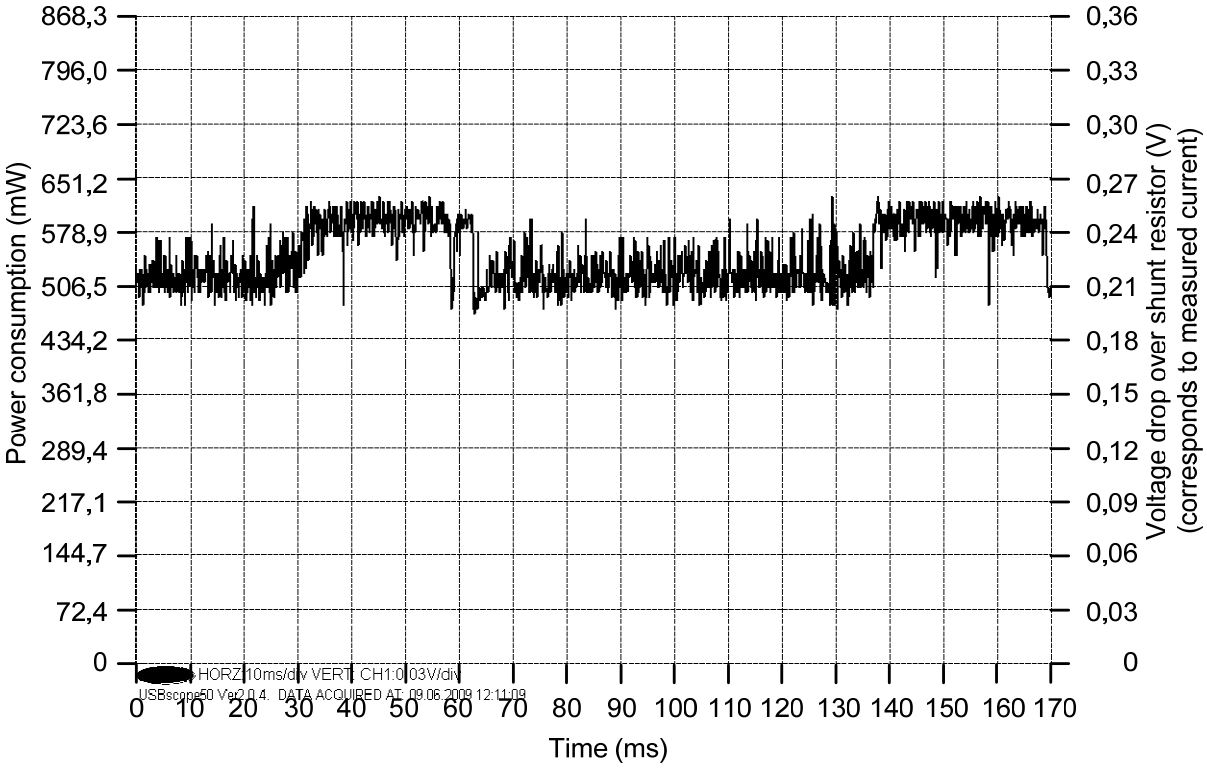



*Figure 29: Screenshot of the oscilloscope during current measurement while decoding one frame on an H.264 video stream*

# 5 Implementation

This chapter describes the implementation of the profiling tool developed within this work. As a target CPU the ARM processor family has been chosen. On one hand because embedded systems are often based on these processors; therefore profiling results for these processors covers a broad range of systems. On the other hand the ARMulator, as described in Section 2.2.1, allows access to inside information, such as cycle counter and memory bus activity and can thus been extended for profiling purposes. The details about the profiling mechanism are described in Section 4.1. This mechanism is not restricted to the ARMulator and has also been applied to other processors, as shown in Section 5.7.

The current implementation is written for the Microsoft Windows operating systems. It requires the ARM Developer Suite or its successor RealView Development Suite, which are described in Section 2.3.1. Within the MEMTRACE source code, operating system specific parts have been separated carefully from the rest of the code. Therefore, porting the command-line implementation of the profiler to other operation systems, e.g. Linux, is possible with modest coding effort.

The simulation speed of ARMulator is approximately 20-50 times slower than the execution on a real processor. The profiler reduces the speed even more, depending on number of profiling features enabled, to a factor of 100-150 times. This is mainly due to the amount of list look-ups required by the profiler. In order to speed up the profiling, specific profiling features can be turned off, for example the bus or instruction profiling. The memory requirements of the MEMTRACE backend go up to 300 MB, depending on the number of traced functions and memory areas found in the program code.

The profiler provides two user interfaces, a graphical user interface (GUI) and a command-line interface (CLI). Both allow access to all features of the profiler and can be controlled by a configuration file. The configuration file is used for storing the lists of functions and variables and for profiling-specific settings, such as page and stack size. The GUI is targeted to quick and easy profiling, whereas the CLI can be used within batch scripts for starting multiple profiling runs, e.g. for achieving results of different cache configurations. The CLI is implemented as a 32-bit command-line application and the GUI uses the Microsoft Foundation Class (MFC) library to create a dialog-based Microsoft Windows 32-bit application.

## 5.1 Workflow

Profiling with MEMTRACE is carried out in three steps. In the first step MEMTRACE analyzes the compiled executable file (axf-image) of the source code and extracts all user-functions. In the second step, the executable file is executed on the ARMulator, in order to perform the profiling. In the third step, the spreadsheet output file is created. The workflow is described in the following by the usage of the CLI. Command-line flags are used for choosing the processing step (initialization, analysis, postprocessing) and the filenames for executable, configuration file, analysis and spreadsheet output file are provided as parameters. The object files for symbol parsing can be specified by the full filenames or by means of wildcards or directory paths for comprising multiple files. Furthermore arguments can be passed to the debugger, e.g. for choosing a processor type or activating the tracer facilities.

The full command-line syntax is:

```
memtrace [-i] [-r] [-x] [-y] [-c configuration-file]
         [-a executable-file] [-p executable-file-parameters]
         [-m output-file] [-e spreadsheet-file]
         [-f spreadsheet-output-format] [-d debugger-options]
         [-t][-?] [-V] [-o object-files]
```

A full description of the command-line parameters is given in Section 8.2.6.

### 5.1.1 Initialization

In the first step MEMTRACE is initialized. It is started by calling:

```
memtrace -i -a executable-file [-c configuration-file]
        [-o object-files]
```

MEMTRACE extracts the functions, global variables and sections from the executable file by comparing the symbols found in the executable with the symbols found in the object files. This step is illustrated in Figure 30. Therefore, all user object-files, libraries and archives of interest should be supplied. The extracted user-functions, variables and sections are written to a configuration file.

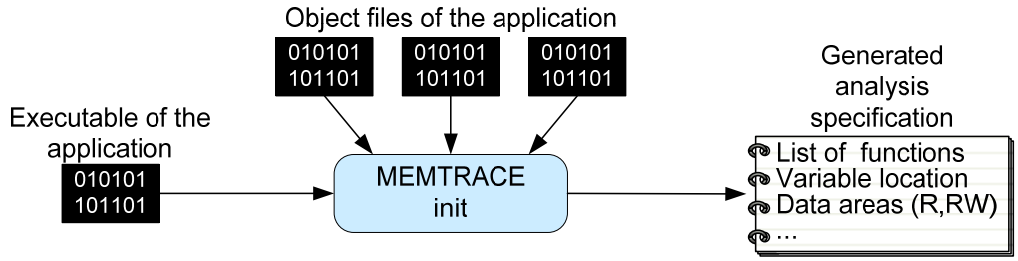



*Figure 30: MEMTRACE initialization step*

This file serves as configuration file for the next processing steps. During the analysis step, this file is used, in order to decide, which functions and memory regions should be traced. Section 4.1.1 describes the process of data acquisition in more detail.

The file can be edited by the user, e.g. for adding or removing functions or defining additional memory areas, such as stack and heap variables, to be traced. The user can define so-called “split functions” by adding “= split” to a specific function in the configuration file. See Section 4.1.1 for the usage of the split mechanism. Additionally the user can control whether the analysis results, e.g. clock cycles, of a function should include the results of a called function (accumulated) or if it should only reflect the function’s own results (self). Typically auxiliary functions, e.g. C standard library functions or simple arithmetic functions are accumulated to the calling functions. For more information on editing the configuration file, see Section 5.5.

### 5.1.2 Analysis

In the second step the performance analysis is carried out, as shown in Figure 31. It is started by executing the following command:

```
memtrace -r -a executable-file [-p exe-file-parameters]
        [-c configuration-file] [-m output-file]
        [-d debugger-options] [-t]
```

The previously generated configuration file defines the functions and variables to be analyzed. Additionally the system parameters, such as the processor type and memory architecture and

timing, can be specified. MEMTRACE connects to the ARMulator via its module interface for the simulation of the user application, as described in Section 4.1. During the simulation the ARMulator provides MEMTRACE with all information required for the analysis. MEMTRACE writes the analysis results for the functions and variables in two separate files. Example results are given in Section 4.1.3. If a “split function” has been specified, these files include separate tables for each call of the "split function". The output files serve as a database for the third step, where user-defined data are extracted from these tables.

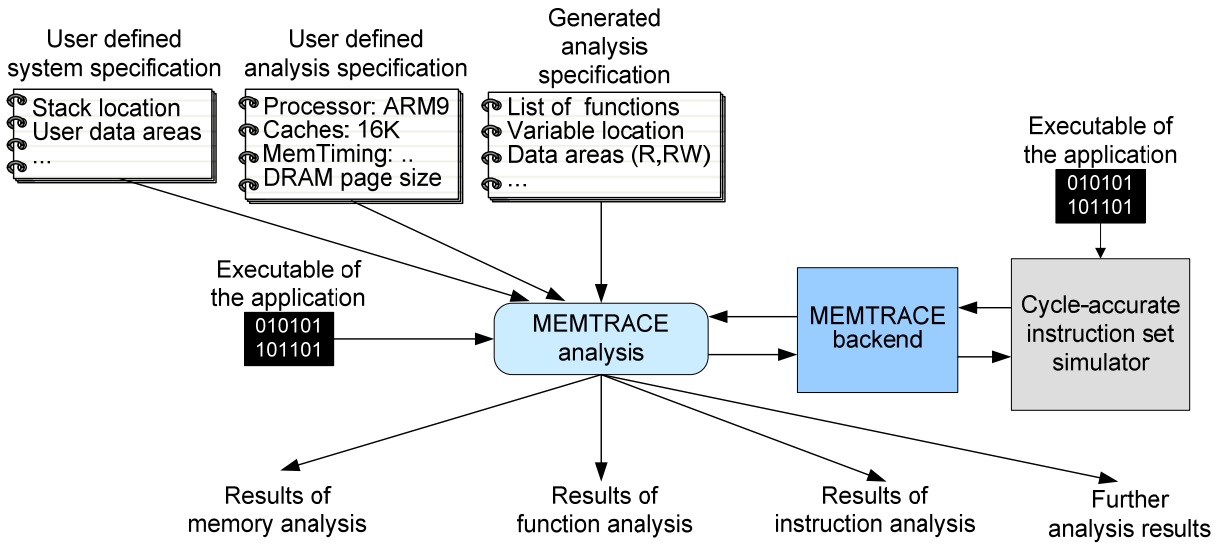



*Figure 31: MEMTRACE analysis step*

### 5.1.3 Postprocessing of the Analysis Results

In the third step user-defined result tables are generated based on the output data produced during the second processing step described above. This step is started by calling:

```
memtrace -x [-f spreadsheet-output-format]
         [-c configuration-file]
         [-m output-file] [-e spreadsheet-output-file]
```

or

```
memtrace -y [-f spreadsheet-output-format]
         [-c configuration-file] [-m output-file]
         [-e spreadsheet-output-file]
```

This step creates a tab-separated file, which includes the user-defined tables, see Figure 32. Before this step can be started, the user-functions and the global variables can be grouped in the `configuration-file`, which has been created in the first step. See Section 5.5 for information on how to group functions and variables. Each group receives the accumulated results of all functions/variables of its members. The grouping can be used to examine the profiling results of a specific software module containing multiple functions. The defined groups are used in the `spreadsheet-output-format` for specifying the format of the output tables. The tables are placed vertically in the output file in the order they are defined in `spreadsheet-output-format`, see Section 5.4.

If no intermediate modification of the configuration file is required, the execution of multiple steps can be combined in one MEMTRACE call.

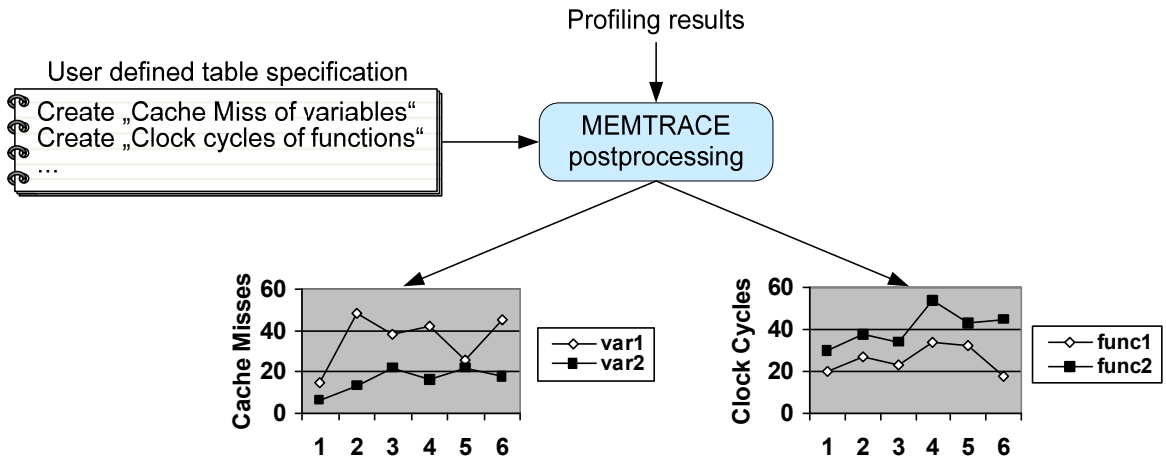



*Figure 32: MEMTRACE postprocessing step*

For example, for running the ARMulator with an executable-file and creating a spreadsheet-output for the function results use:

```
memtrace -r -x -a executable-file [-f spreadsheet-output-format]
        [-m output-file] [-e spreadsheet-output-file]
```

## 5.2 Tool Architecture

The tool is composed of two parts, a frontend and a backend. The frontend acts as an interface to the user, whereas the backend interconnects with the embedded software development suite. The two frontends, GUI and CLI, are based on the same functions library. The backend utilized various tools of the development suite for program information acquisition, e.g. names of all functions in the code, and runtime data acquisition, i.e. profiling data. This structure makes it possible to retarget the tool, on one hand to other processor platforms by exchanging the backend, and on the other to other OS platforms by exchanging the frontend. The entire software is written in C++ and compiled with the Microsoft Visual Studio development environment.

The entire MEMTRACE suite combines the following parts:

- MEMTRACE Base is the base project and provides the command-line interface as well as the backend features for the initialization phase.
- MEMTRACE GUI is the graphical user interface is built on top of MEMTRACE Base.
- MEMTRACE DLL is the backend part for data acquisition during the analysis phase. It is implemented as dynamic link library, which is compiled against the interfaces of the ISS.
- MEMTRACE Coprocessor Interface is a template for coprocessor descriptions within the hardware/software co-profiling environment, as described in Section 4.2.3.
- MEMTRACE Mapfile is the bus interface that extends the basic bus interface of the AR-Mulator to a multi-master bus and adds a DMA controller for data transfer between memory and the coprocessors.
- MEMTRACE Debugger features a minimal debugger for connecting and controlling the MEMTRACE backend with arbitrary ISSes, as described in Section 5.7.

In the following sections the software architecture is shown by describing the most important functions of the MEMTRACE source code. Callgraphs for the different parts of the MEMTRACE suite can be found in Section 8.2.1.

### 5.2.1 MEMTRACE Base

The `main()` function of the MEMTRACE executable performs the following two tasks:

- processing the command-line parameters (`getopt()`)
- starting the functions required for the processing step (init, analysis, postprocessing)

Figure 71 in Section 8.2.1 shows the callgraph of the MEMTRACE executable starting from `main()`. Depending on the processing step, different functions are called, which are described in more detail in the following.

#### 5.2.1.1 Init Step

In the init step, see Section 5.1.1, the MEMTRACE configuration file with the list of project functions is created. The list of project functions is generated by comparing the list of functions found in the executable file (axf-image) with the functions found in the project object files and project libraries. Thus unused functions and non-project library functions can be eliminated. The init step is divided in four steps:

1. `expand_list_of_object_files()` processes the list of object files and paths (provided by the command-line parameter -o), which may contain wildcards and relative paths, and returns a list containing all object files with absolute paths.
2. `create_list_of_project_functions()` creates a list of all project functions (`list_of_project_functions`). For each object file from the list of step 1 a symbol table is created and the function names are extracted from these symbol tables and written to the `list_of_project_functions`.
3. `create_list_of_axf_functions()` creates a list of all functions found in the executable file (`list_of_axf_functions`). This includes project functions and additional library functions.
4. `create_memtrace_ini_file()` creates the configuration file with the list of project functions. The (real) project functions are found by comparing the `list_of_project_functions` with the `list_of_axf_functions`. At first unused functions are eliminated and then local function names, which occur multiple times, are renamed.

#### 5.2.1.2 Analysis Step

In the analysis step, see Section 5.1.2, the actual memory profiling is performed. For this purpose the debugger armsd is started with a specific script file and various options, e.g. for passing parameters to the MEMTRACE DLL. The following three steps are performed:

1. `create_armsd_ini_file()` creates a script file for armsd, which activates additionally the internal ARMulator profiler.
2. `run_memtrace()` initiates the profiling process. It starts armsd with the script file `armsd_run` and the `armsd_options`. The ARMulator, which is called by armsd, executes the executable file `axf_filename` with the command-line parameters `axf_parameters`. Depending on `TracerOn` additionally the internal ARMulator tracer module is turned on. The `axf_filename`, `ini_filename` and the `output_filename` are required in the memtrace_dll and are passed to it via `TARGET`Options.
3. `delete_armsd_ini_files()` deletes the script file generated in step 1.

#### 5.2.1.3 Postprocessing Step

In the postprocessing step, see Section 5.1.3, a tab-separated output file `XLS_FILE` with user-defined tables is generated from the memory profiling results stored in `OUTPUT_FILE`. The table contents are arranged according to the table definitions in `settings` (axf-output-format) and the function grouping in the `INI_FILE`. The processing is divided in five steps:

1. `parse_ini_file()` parses the configuration file in order to create a tree containing all sections (`ini_file_section`).
2. `extract_groups()` extracts all groups of functions defined in the "FunctionList" section and writes them to the list `group_list_template`.
3. `extract_settings()` creates the list of the table definitions (`table_list`) by parsing the `settings` (axf-output-format).
4. `read_memtrace_results()` reads the results of the MEMTRACE profiling from `OUTPUT_FILE` and sorts them into the list of calls (`c_list`) of the split function. After that for each call a `c_list` entry exists, which contains a group list (`group_list_template`) with the results for each group.
5. `print_to_xls_file()` generates the user-defined tables, according to the `table_list`, from data in the `c_list` and write them to the `XLS_FILE`.

### 5.2.2 MEMTRACE Dynamic Link Library (Backend)

The MEMTRACE backend performs the actual memory profiling and is implemented as DLL. It provides six entry functions, which are called from the ARMulator. However, the functions are not directly called from the ARMulator, but via the interface functions defined in tracer_for_memtrace_dll.c. This interface is derived from the original tracer module (tracer.dll). Additionally the Mapfile module (mapfile.dll) is modified for bus tracing, the modules are described in 2.2.1.1 and 2.2.1.2. In Figure 33 the software structure of the MEMTRACE backend is given.

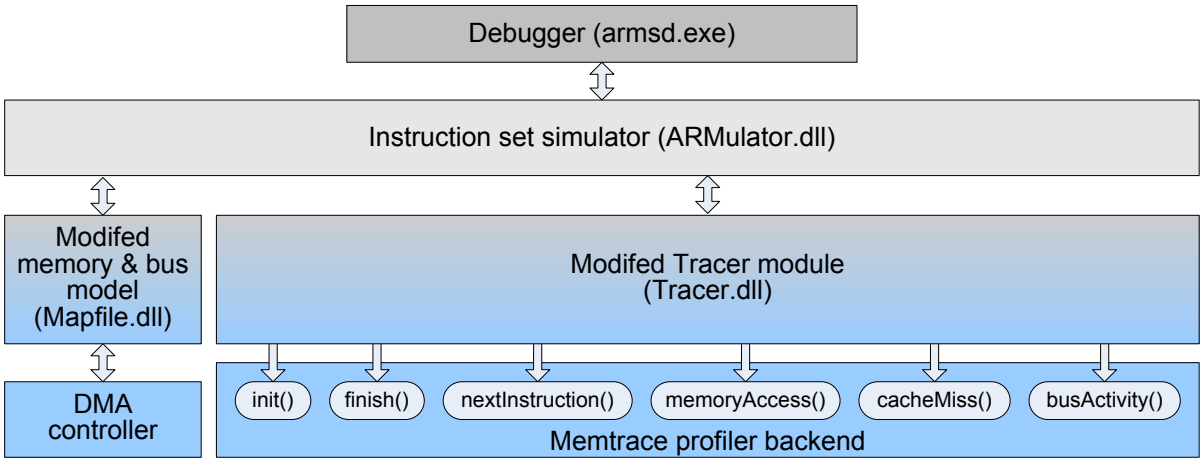



*Figure 33: Software structure of the interface between MEMTRACE backend and ISS*

The Mapfile is extended by a mechanism, which tags each bus cycle with the information, how the bus is currently used. This tag can be either CORE, DMA or IDLE and is identified by the tracer module. The MEMTRACE profiler backend defines a number of entry points, which provide the interconnection between the profiler and the ISS. The Tracer module has been extended by calls to these functions.

The function `init()` is called once when the ARMulator is started and initializes the MEMTRACE profiling. It creates a list of all functions from the symbol table out of the executable file, as described in the initialization step, and marks the user and split functions found in the configuration file. For each function a data structure is created, which contains the function's start address and variables for collecting the analysis results. Similar to the list of functions a list of the variables and other memory areas is created, including the analysis data areas. Finally two pointers, called `currentFunction` and `evaluatedFunction`, are initialized. The first pointer indicates the currently executed function and, if this function should not be evaluated, the second pointer indicates the calling function, to which the result of the current function should be added. If the executable was created in debug mode, i.e. source code information is available, a table for mapping assembly code line to source code line is created.

The function `nextInstruction()` is called by the Tracer module each time the program counter changes. It checks if the program execution has changed from one function to another. If so, the cycle count of the `evaluatedFunction` is recalculated and the call count of the `currentFunction` is incremented. Finally the pointers to the `currentFunction` and `evaluatedFunction` are updated. If `currentFunction` is a split function, the differential results from the last call of the split function up to the current call are printed to the result files. The function is also provided with the disassembly of the instructions. This is used for instruction counting. The disassembly string is parsed in order to identify the instruction and for load/store instructions also the address mode.

`memoryAccess()` is called each time a memory access occurs and increments the memory access counters of the `evaluatedFunction`. Depending on the information provided by the ARMulator, it is decided if a load or store access was performed, and which bit-width (8/16 or 32 bit) was used. Furthermore the ARMulator indicates if a cache miss occurred. Page hits and misses are calculated by comparing the address of the current with the previous memory access and incorporating the page structure of the memory.

If a data cache is available in the processor each time a cache miss occurs the function `cacheMiss()` is called. It accounts the number of data cache misses for the current function and also for the corresponding memory area or variable, which has been accessed.

The function `busActivity()` is called for each bus cycle and provides the MEMTRACE backend with the current bus status tag. Thus, every bus cycle consumed by the `currentFunction` can be classified as CORE or DMA access or IDLE state.

The entry function `finish()` is called when the ARMulator has terminated the simulation. It updates the results of the last `evaluatedFunction` and prints the results of the last call of the split function and the accumulated results to the result file.

## 5.3 Graphical User Interface

The graphical user interface of MEMTRACE allows an easy usage of the tool. It is very useful for the postprocessing task and for novice users of the tool, as most options are provided within a list. The tool is created as dialog-based window with three tabs reflecting the three workflow steps defined in Section 5.1. The GUI is based on the MFC library and built with the Microsoft Visual Studio development environment. It is implemented as a dialog-based 32-bit application. The three processing steps (initialization, analysis, postprocessing) are implemented as three separate tabs in the dialog window. The configurations can be stored in a

configuration file, which is compliant to the CLI configuration file. Thus the user can switch between the two versions of the tool. See Section 8.2.2 for screenshots of the GUI.

## 5.4 Spreadsheet Format Description

The output data to be printed in tables can be specified by the command-line parameter:

```
-f spreadsheet-output-format
```

MEMTRACE creates one output file, which is tab-separated and can be imported by spreadsheet programs. The file may contain multiple result tables. The content of each table is specified in the following format:

```
spreadsheet-output-format: "{<table1>}{<table2>}{<table3>}..."
```

where {<tableX>} is the specification of one output table.

Each table specification {<tableX>} has the following format:

```
{<tableX>}: {table_type;row_column_specifier;row_column_specifier}
```

table_type specifies the data to be displayed in the table. For printing overall results the “ov” type exists and for printing the results of a specific group the “group” identifier is used. The desired group is provided, e.g. “group=myGroup1”. If a table should be created only for one data type, the data type is given together with the table type “data”. The row_column_specifier is supplied for restricting the printed rows or columns. If the row_column_specifier is not given, all results for row and column are printed.

In case of a data table, the groups to be shown can be specified, e.g. “group = MyGroup1, MyGroup2”. Vise versa for group tables the data types are given as a parameter.

Furthermore the row_column_specifier is applied to define a range of results. If the split flag was used, intermediate results have been produced for each call of the split function. Two possible range types are provided for selecting a subset of these results. The simple triple range definition is (<begin>, <end>, <step>). In this step, only the results from "begin" to "end" are printed to the results table, and from these results taking only each "step" intermediate result. The advanced range definition is (<begin>, <end>, <step>, <period>, <period_end>). Here the results from "begin" to "end" taking each "step" call are taken. This is periodically repeated in a distance of "period". The repetition is finished when "period_end" is reached. For example, for printing results of the calls: 3, 4, 5, 9, 10, 11, 15, 16, 17 specify (3, 5, 1, 6, 17).

The available data types are listed in Table 13. If these types are used in <table_type>, the <data_type> can be extended by a "+" (such as CY+) for printing an extra row with the accumulated result. The accumulated results are only the sum of the visible groups, not of all groups.

*Table 13: Data types for table results*

| Flag | Function |
|---|---|
| **LS** | Add "all load store" column to table |
| **LD** | Add "all load" column to table |
| **L8** | Add "load_8_16" column to table |
| **ST** | Add "store" column to table |
| **S8** | Add "store_8_16" column to table |
| **PH** | Add "page hit" column to table |
| **PM** | Add "page miss" column to table |
| **CM** | Add "cache miss" column to table |
| **LS** | Add "all load store" column to table |

A typical `spreadsheet-output-format` may look like:

```
"{data=CY+;group=control,inverse
scan;(2,10,4)}{group=control;data=LS,PH,PM;(30,50,10)}{OV}"
```

This would print the following three tables. The first table (Table 14) represents the cycle counts for the groups “control” and “inverse scan” for call 2, 6 and 10 next to an extra column for the accumulated results. The second table (Table 15) for the group “control” shows load, store, and page hit and miss for calls 30, 40 and 50. The last table contains the overall results (Table 16).

*Table 14: Data types for table results (cycle results)*

| Frame | Control | Inverse scan | Sum |
|---|---|---|---|
| **2** | 419 | 0 | 419 |
| **6** | 2492718 | 56471 | 2549189 |
| **10** | 3672817 | 82832 | 3755649 |

*Table 15: Data types for table results (group results)*

| Frame | All load store | Page hit | Page miss |
|---|---|---|---|
| **30** | 279096 | 79693 | 199403 |
| **40** | 378873 | 107272 | 271601 |
| **50** | 406627 | 116077 | 290550 |

*Table 16: Data types for table results (overall results)*

| Group | Cycles | All load store | Load | Load 8/16 | Store | Store 8/16 | Page hit | … |
|---|---|---|---|---|---|---|---|---|
| **Control** | 3182569 | 423001 | 230185 | 275 | 192816 | 96597 | 122280 | … |
| **Inverse scan** | 70627 | 14642 | 8946 | 880 | 5696 | 0 | 9260 | … |
| **Inverse transformation** | 373421 | 62966 | 37407 | 0 | 25559 | 9488 | 38455 | … |
| **Entropy decoding** | 1398896 | 277109 | 188338 | 822 | 88771 | 956 | 176510 | … |
| **…** | … | … | … | … | … | … | … | … |

## 5.5 The Configuration File

The MEMTRACE configuration file contains the list of functions, the list of variables and some further settings. The file is generated in the first step, the initialization step, see Section 5.1.1. It can be edited by the user, in order to control the second (analysis step) and the third step (postprocessing step). The most common changes made on this file, is the setting of a “split function” and the grouping of functions and variables.

### 5.5.1 File Format

The file format of the configuration file is similar to the ARM configuration file format as used and defined within the RealView Development Suite [14]. Table 17 shows a list of the syntax elements.

*Table 17: Syntax elements of the configuration file*

| Syntax | Description |
|---|---|
| **MyTag** | Defines a tag |
| **MyTag2 = Value1** | Defines a tag and sets it to a value |
| **MyTag2 = Value1 Value2** | Defines a tag and sets two values for it |
| **;; comment line** | Comment line for description |
| **; commented-out line** | Comment line used for commenting out |
| **{ MySection = SectionName** | Beginning of a section of type MySection named SectionName |
| **SectionTag1** | Definition of a tag inside a section |
| **}** | End of a section |

### 5.5.2 List of Functions

All functions and groups of functions are defined as tags inside the section of type and name "FunctionList". One specific function can be a split function, see Section 5.1.1. This is indicated by adding “ = split” after the function name. Functions can be grouped in sections of type “group” supplied with a user-defined name. This name is later used in the `spread-`

sheet-output-format, see Section 5.4. The groups are placed inside the group “FunctionList”. An example of this format is:

```
{ FunctionList = FunctionList
  { group = MyGroup1
    function1
    function2
  }
  { group = MyGroup2
    function3 = split
    function4
  }
}
```

*Listing 11: Definition of functions in the configuration file*

### 5.5.3 List of Variables

A list of global variables and user-defined memory areas is defined similar to the list of functions. The list is divided into three parts, the list of global variables the list of sections and the list of fixed memory areas. The first two lists are automatically created in the initialization step of MEMTRACE. The third one is defined by the user. An example of this format is:

```
{ MemoryMap = MemoryMap
  { FixedAreaList = FixedAreaList
    myStack = 0x7FFE000 8192
    HeapVariable1 = 0x000AB95C 25344
    HeapVariable2 = 0x000B1C5C 4
  }
  { SectionList = SectionList
   'ER_RO'
   'ER_RW'
   'ER_ZI'
  }
  { GlobalVariableList = GlobalVariableList
    GlobalVariable1
    GlobalVariable2
  }
}
```

*Listing 12: Definition of variables in the configuration file*

The GlobalVariableList includes all global variables, which are defined in the user application. The SectionList includes the memory regions (sections) of the user application. This sections where either automatically defined by the linker (ER_RO, ER_RW, ER_ZI) or by the user in a scatter-loading file. The address and size of the global variables and the sections are retrieved from the compiled executable file of the source code during the analysis step. In the FixedAreaList additional memory areas are defined for profiling. These areas must be specified with its start address and size:

```
<name> = <startaddress> <size>
```

Typical memory areas include the internal or external memory space, stack, heap or heap variable. However, as the start address and size of some areas might change after a recompilation of the application, they need to be refreshed. This problem may occur with the heap or heap variables. Therefore the start address and sizes need to be verified during program execution, e.g. with printf() of the malloc addresses.

### 5.5.4 Global Settings

By including the additional section `Global`, various settings can be given to control MEMTRACE. For example, the tags `BaseAddr` and `PageSize` of data memory can be applied for page hit/miss calculation. The `BaseAddr` specifies the base addr of the first page in memory. It needs to be expressed as hexadecimal value (with leading 0x). The `PageSize` tag defines the size of each page in bytes. `PageSize` need to be expressed as integer value. The base address (`StackBaseAddr`) and size (`StackSize`) of the stack are defined similarly.

The global sections format is:

```
{ Global = Global
  BaseAddr      = 0x0
  PageSize      = 128
  StackBaseAddr = 0x08000000
  StackSize     = 8192
}
```

*Listing 13: Definition of global settings in the configuration file*

## 5.6 Infrastructure for System Architecture Profiling

Beside the previously described software related profiling MEMTRACE also supports a high-level hardware/software co-profiling. For this purpose the bus model of the ISS has been extended to multi-master bus. Additionally to the memory devices, coprocessor models can be connected via memory-mapped interface and a DMA controller.

### 5.6.1 Hardware/Software Cosimulation Interface

The coprocessor interface, shown in Figure 34, supplies the interconnection of a coprocessor to the AHB.

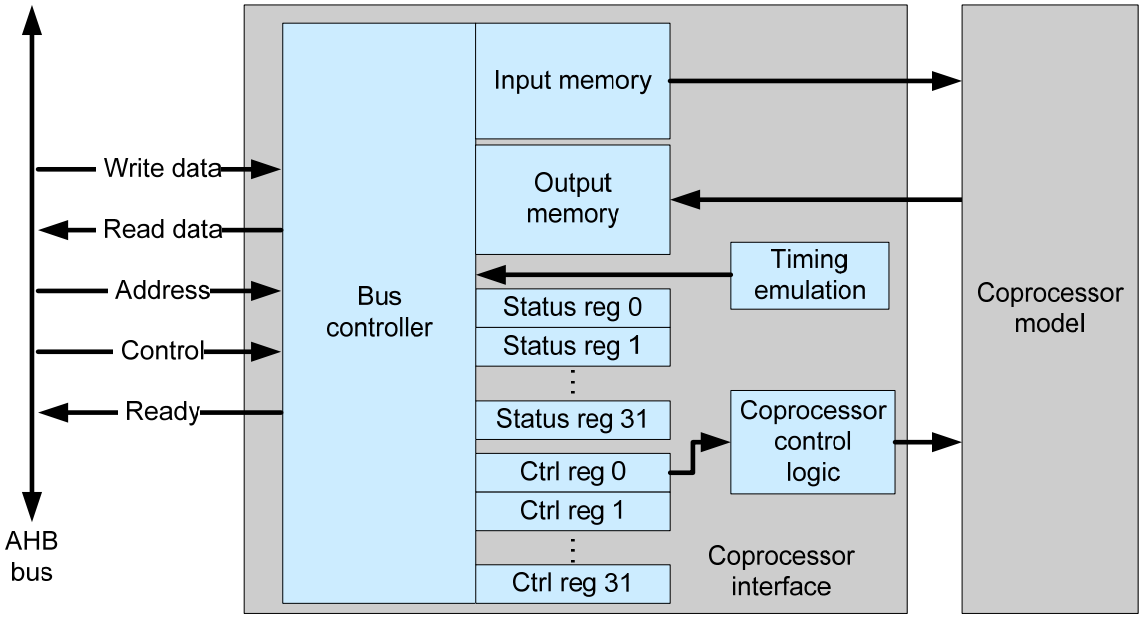



*Figure 34: Connecting coprocessors to the AHB of the ISS*

The interface supports the following features:

- bus slave controller for the AHB including local address decode
- set of predefined 32-bit status and control registers
- separate byte-addressable input and output memory units
- control logic for starting and stopping the coprocessor model
- simple timing emulation for simulating the computation delay of the coprocessor
- easy setting of parameters, such the memory map register file size and timing parameters

The parameters can be set in a text file and modified after compiling the coprocessor model, which supports the design space exploration by evaluating different hardware setups and timing. Coprocessor models are written in the C language and connected to the interface by a call of the entry point function. More details of the coprocessors can be found in Section 4.2.4.

### 5.6.2 DMA Controller

The data transfers into hardware accelerators or other bus components have a tremendous influence on the overall performance. For efficiently outsourcing this data transfer task DMA controllers are applied. The MEMTRACE hardware profiling environment includes a highly efficient DMA-Controller with the following features:

- multi-channel (parameterizable number of channels)
- 1D- and 2D- transfers
- activation FIFO (non-blocking transfer, autonomous)
- internal memory for temporary storage between read and write
- burst transfer mode

Thus the designer is enabled to determine the influence of different DMA modes in order to find an appropriate trade-off between DMA Controller complexity and required CPU activity. The DMA controller is embedded in the memory architectures as additional bus master component, see Section 4.2. A software API allows configuration and activation of the DMA controller and returns information on the current status of the data transfers.

## 5.7 Retargeting to Other Embedded Processors

The previous sections describe the implementation of the profiling tool in conjunction with the ARMulator for profiling ARM processors. The simple interface of the MEMTRACE backend, as described in Section 5.2.2, allows an easy retargeting of the basic profiling features to other processor simulators or emulators. In order to allow the profiling of other processors, the profiler needs access to the tracing information of these processors, which includes the following basic parameters:

- cycle counter
- program counter
- activity on the data and instruction bus including transferred data and addresses

Optionally, information about cache miss counts and instruction disassembly can be evaluated for further profiling details. This information needs to be on a cycle-accurate basis and can be provided either by the trace buffer of a hardware emulator, such as an in-circuit emulator or

an FPGA implementation, or by an ISS. The actual retargeting process is implemented by connecting the MEMTRACE backend interfaces, described in Section 5.2.2, to the simulator or debugger. The debugger needs to call the interface functions listed in Table 18 at the appropriate events and provide them with the required information.

*Table 18: Interface callback functions of the MEMTRACE backend and their activation events and input parameters*

| Callback function | Activation event | Input parameters |
|---|---|---|
| **Init() *** | Start of program execution | Current simulation time*<br>Current program counter* |
| **Finish()*** | End of program execution | Current simulation time* |
| **NextInstruction()*** | Every instruction | Current simulation time*<br>Current program counter*<br>Disassembly string of the instruction<br>A flag, if the instruction is executed<br>Instruction binary format (only for energy) |
| **MemoryAccess()** | Every memory access | Accessed memory address*<br>Load/store flag<br>Size flag<br>Transferred data word (only for energy) |
| **CacheMiss()** | Every cache miss | Accessed memory address* |
| **BusActivity()** | Every bus clock cycle | Bus master id |

For a basic retargeting, only the *init()*, *finish()* and *nextInstruction()* interfaces need to be connected to the simulator. These functions should be instantiated as callback functions of the debugger/simulator. This means the debugger should call these functions each time the specified event occurs and provide them with the mandatory parameters (marked with a star), as well as with the optional information given in the table above if available. These three interfaces allow a clock cycle profiling of the processor on a function-accurate level. For additional coverage of memory access within the analysis, the *memoryAccess()* interface needs to be connected to the simulator. If the processor has a cache infrastructure, the debugger can be connected to the *cacheMiss()* interface in order to provide cache miss profiling. Similarly, the bus activity can be analyzed with the appropriate interface function.

A full retargeting of MEMTRACE also includes the profiling of instruction usage and energy consumption. Changes in the MEMTRACE source code are required to support these features. The changes for the instruction set mainly include the code for parsing and interpreting the disassembly format of the processor, e.g. extracting the instruction name, used registers and address mode. The processor-specific parts of the energy model are covered by the equations for energy calculation and the power consumption tables for each instruction class, as described in Section 4.5.4. The existing implementation of both features for the ARM processor is comprised of 500 lines of code, which can be taken as a template for the retargeting. The complexity of this process depends on processor architecture; for a simple RISC architecture the effort can be estimated as low, as the ARM processors use a typical RISC instruction set.

Besides the connection to the ISS, the MEMTRACE backend also needs to extract the symbol table from the file in order to map the functions to the program memory address space. In the

current implementation, MEMTRACE applies a tool called fromelf (see Section 2.3.1), which allows the symbol table of any ELF compatible object or executable file, a widely used linker file format, to be extracted. If a different format is used, the MEMTRACE symbol table parser needs to be adapted. In the current implementation it consists of less than 150 line of source code.

The ISSes are usually provided either by the processor vendor or by third parties. If simulators are not available for given processor, they can be created either manually or automatically from a higher-level description. Verilator, as described in Section 2.2.2, creates C++ simulation models from a Verilog description of a processor. If the processor is not available as Verilog HDL model, it can be described in an even higher abstraction level, for example with the LISA language [76]. The CoWare Processor Designer [29] can be used to create HDL models and compiler tools for processors described in LISA.

In order to prove the feasibility of the retargeting procedure, a toolflow incorporating CoWare Processor Designer, the Verilator and the MEMTRACE backend has been created. Thus MEMTRACE can profile processors available either as Verilog HDL or as LISA descriptions. The section below describes the retargeting process and depicts the minimal effort it requires.

### 5.7.1 Toolflow for Profiling LISA and Verilog Processor Models

The toolchain given in Figure 35 has been developed to use MEMTRACE to profile processor described in a high-level description language.

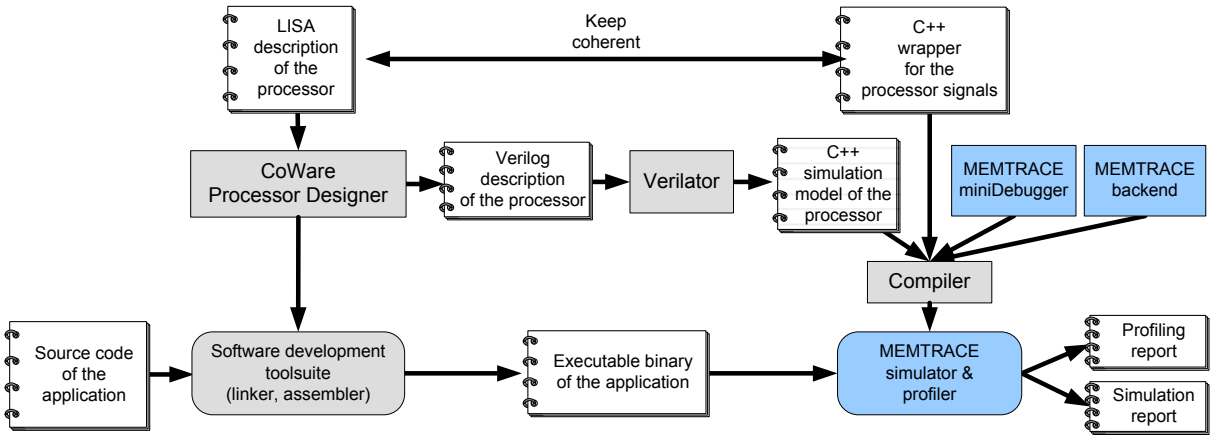



*Figure 35: Profiling toolflow incorporating CoWare Processor Designer, Verilator and MEMTRACE*

The design procedure starts with a processor description in the LISA language. This description is processed by the CoWare Processor Designer to generate a Verilog description of the processor and the required software development tools, such as an assembler and a linker. The Verilog description generated is further processed by the Verilator to generate a C++ simulation model. This model is then compiled with the MEMTRACE backend and the miniDebugger libraries to form the combined simulator, debugger and profiler. In order to ease the retargeting process, a generic interface between the simulation model and the debugger is defined. Therefore, the simulation model needs to be enclosed by a wrapper mapping the processor signals to the MEMTRACE backend interfaces functions. Figure 36 illustrates this interconnection in more detail.

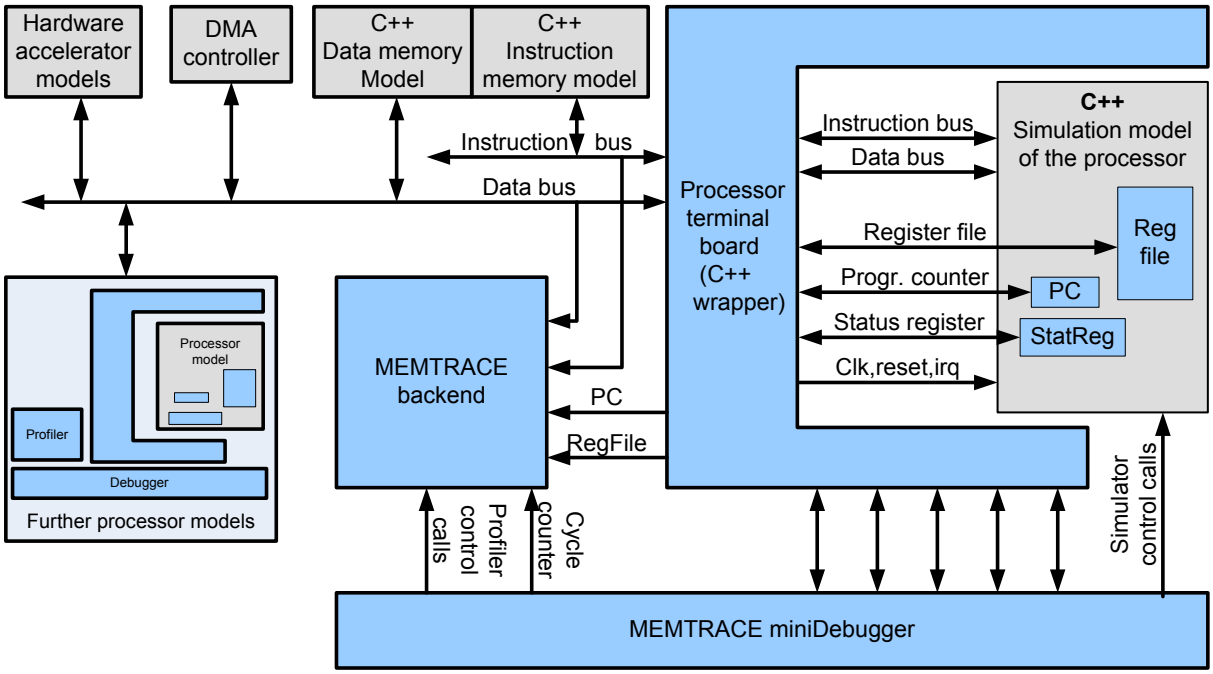



*Figure 36: Interconnection of the C++ processor model with the MEMTRACE backend and miniDebugger and with further system components by means of a terminal board*

The right side shows the C++ simulation model generated by the Verilator. The model consists of a C++ source and a header file containing a class definition that corresponds to the Verilog description. The input and output ports of the top level module are converted to member variables with the same name. Ports and signals of lower level modules are visible, in the format:

lowLevelModuleName __DOT__lowLevelModuleName__DOT__signalName

Such names are dependent on the signal and port naming within the Verilog processor model, so the wrapper, which acts a terminal board, is used to provide a generic interface to other system and simulation modules.

The interface provides access to the most common and important parts of the processor, including the instruction and data busses, register file, program counter and status register, as well as the control signals for clock, reset and interrupts. Thus when replacing the processor model with a different model, only the following needs to be configured:

- connection between the terminal board and the processor model
- bit-width and timing of the busses
- number of registers

Listing 14 shows a sample implementation of the terminal board as it needs to be configured by the designer in order to suit the specific processor. The initial for-loop connects all registers from the processor model (found in the signal array “REG_R” in low-level module “RFile”) to the generic register array “Reg”. Similarly the program counter (“PC”) and the control (“CPSR”) and status (“SPSR”) registers are connected. The interrupt (“fiq”,”irq”) and wait (“wait”) signals are not used in this processor, therefore they are set to a constant zero value. The clock (“clk”) and reset (“reset”) signals are connected to the corresponding top-level signals of the processor model, similar to the data and instruction busses. All these connections between the generic and processor-specific variables are established as pointers, and

therefore allow bi-directional access, i.e. the generic variables can be used for reading and writing the processor-specific variables.

```
void setSignals(Vtop *top) {
 for (int i=0;i<NUMREGS;i++)
 {
   Reg[i]  = top->v__DOT__RFile__DOT__REG_R[i]);
 }
 PC    = top->v__DOT__pipe__DOT__DC_EX__DOT__DC_EX_p_pc_internal;
 CPSR  = top->v__DOT__RegFile__DOT__REG_PSR;
 SPSR  = top->v__DOT__RegFile__DOT__REG_PSR;
 fiq   = &constBit0;
 irq   = &constBit0;
 wait  = &constBit0;
 clk   = top->clk_main;
 reset = top->rst_main;
 DataWriteBus       = top->data_mem_data_in_wp0;
 DataWriteBusAddr   = top->data_mem_wr_addr_wp0;
 DataReadBus        = top->data_mem_data_out_rp0;
 DataReadBusAddr    = top->data_mem_rd_addr_rp0;
 InstrBus           = top->prog_mem_data_out_rp0;
 InstrBusAddr       = top->prog_mem_rd_addr_rp0;
 WriteEnable        = top->data_mem_ew_wp0;
 ReadEnable         = top->data_mem_rd_enab;
 Byte1Word0         = &dummyBit0;
 MemoryAccess1      = &dummyBit0;
 data_ready         = top->data_mem_ready;
}
```

*Listing 14: Implementation of the processor terminal board (C++ wrapper)*

The mandatory system extension is a model of the data and instruction memory connected to the system bus. The technique and components for hardware/software cosimulation described in Section 4.2.3 can also be applied here. Multi-processor systems can be generated by adding further processor models including their debuggers and profilers, as shown in Figure 40 on the left side. Even hierarchical bus systems can be created within this environment.

The processor simulation is controlled by a debugger. The rudimentary MEMTRACE miniDebugger allows running and stepping through the assembly code and viewing of register and memory values. The debugger also controls the MEMTRACE profiler backend. The required interconnection to the processor internals is provided by the terminal board. The debugger could also be replaced by a full-featured debugger, for example by means of a debugger plugin to the Eclipse software development platform [101].

The simulation environment described here allows a simple retargeting of the profiler to any processor that is available as a C source code model. The Verilator extends the supported processor range to Verilog models. The profiling speed has been measured as 50,000 simulated processor cycles per second running on a 3.6 GHz Intel Xeon PC.

The design flow was tested by the example of a simple RISC processor, similar to the SPARC architecture, developed by the Embedded System Group at the Fraunhofer Heinrich-Hertz-Institut (HHI). The LISA description contains about 5500 lines of simple and well-structured code. Equivalent HDL descriptions are far more complex. The description generated by the Verilator, for example, contains 25,000 lines of code. Thus the LISA language allows an easy

description and modification of a processor model, and the results delivered by MEMTRACE support a profiling-based exploration of these design alternatives.

Table 19 shows some example results from profiling the simple RISC core. This retargeting has been performed for instructions and memory functions. Thus the results include the calls and cycle counts, access statistics for the memory and the stack, as well as page hit and miss counts for each function.

*Table 19: Example results for profiling a program running on a simple RISC processor*

| Group | Calls | Cycles | All load/store | Load | Store | Page hit | Page miss | Stack read | Stack write |
|---|---|---|---|---|---|---|---|---|---|
| **Main** | 1 | 36 | 30 | 0 | 30 | 29 | 1 | 90 | 90 |
| **Func1** | 1 | 37 | 30 | 0 | 30 | 30 | 0 | 0 | 30 |
| **Func2** | 1 | 437 | 386 | 179 | 207 | 385 | 1 | 0 | 30 |
| **Sum** | | 510 | 446 | 179 | 267 | 444 | 2 | 90 | 150 |

## 5.8 Power Measurement Setup

For the generation of the power consumption model a setup has been created for measuring the dynamic current flowing through the embedded system under test. As described in Section 4.5, an Altera Excalibur device, namely the EXPA1, has been chosen for this purpose. Altera provides a development board for the device, which uses the Texas Instruments PT 6983C [95] as power supply. The PT 6983C is a switching regulator for dual output voltage. From a 12 V input voltage it generates 1.8 V and 3.3 V output. It is manufactured on a separate PCB, which is connected by 23-pins to the development boards. This layout makes the power measurement easier, as the pins can be cut, to infer a current meter. For the current measurement a shunt resistor has been used, as described in Section 4.5.2. The voltage drop over the shunt resistor is amplified by an instrumentation amplifier and measured by a digital oscilloscope. A picture of the entire measurement setup is given in Figure 37.

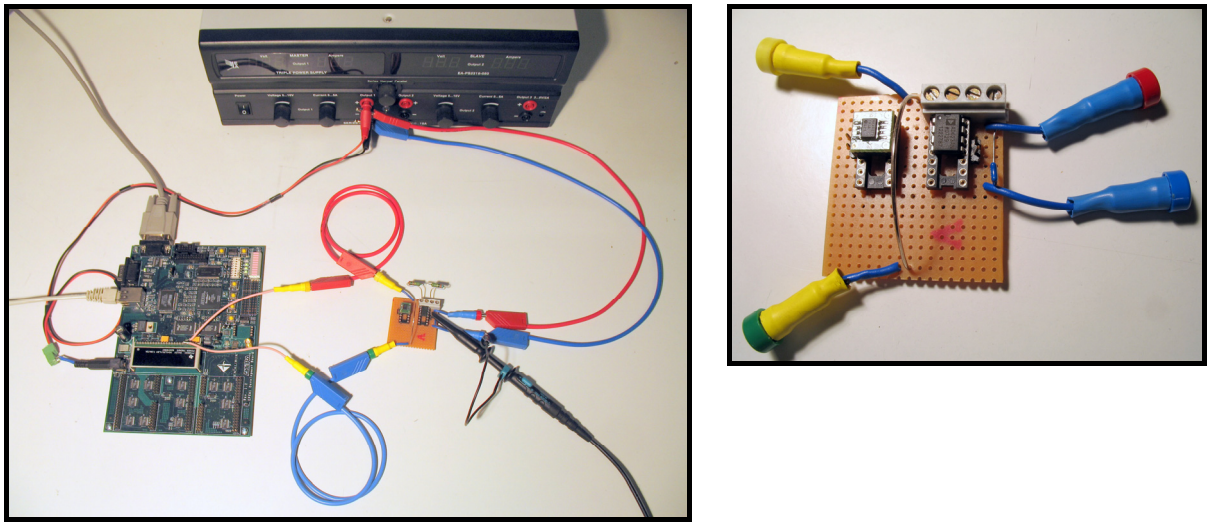


*Figure 37: left: Setup with FPGA board, measurement board, power supply and oscilloscope probe; right: measurement board with shunt resistor and the two amplifier*

### 5.8.1 Calibration of the Measurement Setup

All components of the measurement setup need to be calibrated; respectively their error ranges need to be considered. The accuracy of the sense resistor is of high importance, as it has a linear influence on the measurement accuracy. As a resistor either a discrete device or a resistor wire with the correct length can be chosen. In both cases, the setup needs to be calibrated by measuring the resulting value. For the measurement of a very low resistor value, the voltage-correct measurement of the resistor needs to be chosen, as voltage meters have an input impedance in the range of megaohm whereas for current meters it is only a few ohm. For this measurement the accuracy of the voltage meter also needs be considered. Table 20 shows accuracy for the applied measuring instrument, a METEX MXD-4660A for current measurement and a FLUKE 27 for voltage measurement.

*Table 20: Accuracy of the measurement instruments*

| Measuring instrument | Range | Accuracy | Minimum value |
|---|---|---|---|
| **Metex MXD 466A** | 0 - 200 mA | ±(0.5 % + 3 digits) | 10 µA |
| **Fluke 27** | 0 - 320 mV | ±(0.1 % + 1digit) | 100 µV |

A shunt resistor with 100 mΩ has been chosen, the accuracy is given with 5 %. For achieving a high accuracy during the resistor measurement, a current value close to the available maximum range of 200 mA is chosen. The following values have been measured:

$I = 193.82\ mA$

$V = 19.5\ mV$

The resistor value can be calculated by:

$$R = \frac{V}{I} \tag{21}$$

Considering the accuracy of the instruments this becomes:

$$R = \frac{V_{measured} \pm V_{err}}{I_{measured} \pm I_{err}} = \frac{V_{measured} \pm (V_{measured} \cdot 0.001 + 1 \cdot V_{lastDigit})}{I_{measured} \pm (I_{measured} \cdot 0.005 + 3 \cdot I_{lastDigit})} \tag{22}$$

Thus the range for the resistor value is $R_{min}$ to $R_{max}$:

$$R_{max} = \frac{V_{measured} \pm V_{err}}{I_{measured} \pm I_{err}} = \frac{V_{measured} + (V_{measured} \cdot 0.001 + 1 \cdot V_{lastDigit})}{I_{measured} - (I_{measured} \cdot 0.005 + 3 \cdot I_{lastDigit})} \tag{23}$$

$$R_{min} = \frac{V_{measured} \pm V_{err}}{I_{measured} \pm I_{err}} = \frac{V_{measured} + (V_{measured} \cdot 0.001 + 1 \cdot V_{lastDigit})}{I_{measured} - (I_{measured} \cdot 0.005 + 3 \cdot I_{lastDigit})} \tag{24}$$

which becomes

$$R_{max} = \frac{19.5mV + (19.5mV \cdot 0.001 + 1 \cdot 100\mu V)}{193.82mA - (193.82mA \cdot 0.005 + 3 \cdot 10\mu A)} = \frac{19.6195}{192.8209} = 101.74986\ m\Omega \tag{25}$$

$$R_{min} = \frac{19.5mV - (19.5mV \cdot 0.001 + 1 \cdot 100\mu V)}{193.82mA + (193.82mA \cdot 0.005 + 3 \cdot 10\mu A)} = \frac{19.3805}{194.8191} = 99.47947\ m\Omega \tag{26}$$

The estimated value for the resistor is then:

$$R_{mean} = \frac{R_{max} + R_{min}}{2} = 100{,}61\,m\Omega \tag{27}$$

with an error of:

$$R_{error} = \pm \frac{(R_{max} - R_{min})/2}{R_{mean}} \cdot 100 = \pm \frac{R_{max} - R_{min}}{R_{max} + R_{min}} \cdot 100 = \pm 1{,}13\% \tag{28}$$

With this measurement the initial 5 % tolerance of the resistance value can be reduced to 1.13 %. When calibrating a resistor wire to a specific the same method can be applied. Furthermore the wire also allows to adjust the resistor value exactly to a required value, by either reducing the length of the wire (reduction of the resistance) or by using a rasp to decreasing the diameter at a specific position (increase the resistance).

Furthermore it is also important to consider the temperature dependency of the resistance, either caused by a change of the environmental temperature or due to the power dissipation of the resistor itself. Usually the latter is the more frequent reason, especially when for higher power dissipation, which comes close to the maximum specified value for the resistor. Also parasitic inductance and capacitance need to be considered.

Especially wire-wound resistors, which have a similar design as coils, show a high inductance (> 100 nH), and therefore significantly influence the impedance of the resistor already for frequencies in the range of hundreds of kilohertz.

The accuracy of the oscilloscope, a Yokogawa DL9710L, is given as

1.5 % *of* 8 *div* + *offset voltage accuracy*

with an offset voltage accuracy of:

1 % *of offset setting* + x *mV.*

For the voltage range expected during the measurement, which is below 400 mV, the offset component *x* is set to 0.2 mV.

### 5.8.2 Software Test Suite

A set of 90 assembly code sequences has been developed, which reflect the different energy contribution described in Section 4.5.3. For achieving a constant current value, the instructions have been executed in a loop. In order to decrease the influence of the loop overhead, i.e. counter update and check, jump instruction and pipeline stalls, loop unrolling to a long instructions sequence should be performed. On the other hand, the sequence length should not exceed the instructions cache size for omitting cache misses during the measurement. A loop body of around 1000 instructions has been chosen that leads to 4 kB of code and 10 cycles for loop handling, which in turn leads to a loop handling influence of around 1 %. The power consumption respectively the current has been measured for each for each code sequence. As the power consumption can be considered as stable during the execution of a specific instruction sequence, a DC measurement setup can be used. Thus, during the test scenario, the current can be evaluated directly with a current meter, which leads to an increased accuracy as compared to the measurement methods described above, which is important for this basic model features.

In a first test suite the intra energy consumption for different instructions has been measured. The results, see Section 8.2.3, show that the instructions can be grouped in classes of instructions with similar power consumption, for example all simple arithmetical instructions, such

as ADD, SUB and CMP have the same power consumption. The instruction following instruction classes have been identified:

- arithmetic instructions (ADD, SUB, CMP,…)
- logical instructions (AND, EOR, …)
- move instructions (MOV, MVN, NOP,…)
- multiply instructions (MUL, MLA,..)
- branch instructions (B, BL, BLX)
- load/store instructions (LDR, STR)
- multiple load/store instructions (LDM, STM)

The NOP instruction is a pseudo-instruction, which actually is transformed to “mov r0, r0”. For these classes a detailed model has been created. Not covered by this model are instructions, which are specific to the processor implementation and are not available within every processor, such as:

- coprocessor instructions
- floating point instructions
- Thumb instruction set
- other miscellaneous instructions (SWI, BKPT)

The modeling of floating point instructions depends on their implementation, either as an emulator or as a coprocessor. If a coprocessor is used, the coprocessor should be modeled separately in order to achieve an accurate model. This is generally the case with coprocessor instructions, as their power consumption is highly dependent on the coprocessor. Blume et al. [24] create such a separated model for the floating point coprocessor of an ARM processor. This model is part of a power model based on their hybrid FLPA/ILPA technique described in Section 2.3.4.4. Similar to the approach for the CPU taken in this work, the instructions of the coprocessor are combined into groups. As the processor used in this work does not provide any arithmetic coprocessors, there is no need to create such a coprocessor power model here.

The influence of the MMU, which is part of the coprocessor CP15, during load, store and instruction fetch operations, is modeled as part of the appropriate operations. Other accesses to the coprocessor CP15 occur very seldom, thus their influence can often be ignored. The same is true for miscellaneous other instructions, such as software interrupts (SWI) or breakpoints (BKPT). The Thumb instruction set is a specific feature of the ARM architecture. This is considered to be out of the scope of this work, as the methodology presented here should cover only generic features of RISC architectures.

# 6 Application of the Profiler

The profiling tool has been applied during the design of several software applications as well as an embedded system architecture. The tool has proven its applicability for information gathering and software optimization as well as for developing a complex system architecture with a RISC processor, multiple memory devices, busses and hardware accelerators. The following sections describe two examples for usage of the profiler within the analysis and design of embedded hardware/software systems.

## 6.1 H.264/AVC Decoder Profiling

In this study the MEMTRACE profiler is used for evaluating the feasibility of reaching the performance requirements for processor-based DVB-H [33] system. DVB-H is a standard for broadcasting of digital audio and video content to mobile devices. The content is encoded using highly efficient compression methods, namely AAC-HE for audio data and the H.264/AVC codec (see Section 3.4.1 provides more detailed information about the H.264/AVC video coding standard) for video content. DVB-H focuses on high mobility and low power consumption of the receivers. The most demanding part of the receiver in terms of computational requirements is the H.264/AVC video decoder. Therefore a detailed profiling of the H.264/AVC video decoder is performed. DVB-H defines different so-called capability classes, which determine the image resolution and framerate. Here, the capability class B has been chosen, which defines that the decoder need to compliant to level 1.2 of the H.264/AVC standard, with a maximum a resolution of 352x288 pixels (CIF) and a framerate of 15 fps.

The target device is an SoC based on an ARM11 processor. By using the advanced instruction set of the ARM11 and by the use of on-chip resources it is potentially possible to optimize the software. The aim of the optimization is to meet the relevant performance targets for the system with the minimum level of utilization of CPU bandwidth. The H.264/AVC decoding software is analyzed to establish what optimizations would be beneficial and to quantify the potential improvements. The techniques analyzed are:

- algorithmic optimizations
- assembly coding of critical code segments
- usage of ARM11 SIMD instructions
- usage of ARM11 TCMs and DMA

### 6.1.1 Description of the Test Scenario

The H.264/AVC decoder software has been optimized (using only compiler optimizations) for the ARM processor architecture. Three system specifications were evaluated. The systems consist of a processor with separate instruction and data caches and memory, see Figure 38. As can be seen, instead of the ARM11 core, an ARM9E model has been used for the profiling, due to the fact that when the case study was carried out, an ARM11 simulator was not available. From the available processor models, the ARM9E was the most similar core and its similarities to the ARM11 core were considered to be sufficient for the requirements of this case study. The influence of the SIMD instructions additionally available in the ARM11 core is estimated in Section 6.1.3.1.

The first system uses a fast SRAM, whereas the second and the third system use slower DRAM. In all systems all data and instructions are stored in the memory.

The test systems have been specified with processor parameters defined in Table 21 and the three different memory models listed in Table 22. For the SRAM configuration a very fast memory model with zero wait states has been defined and for the DRAM configurations two distinct models are used.

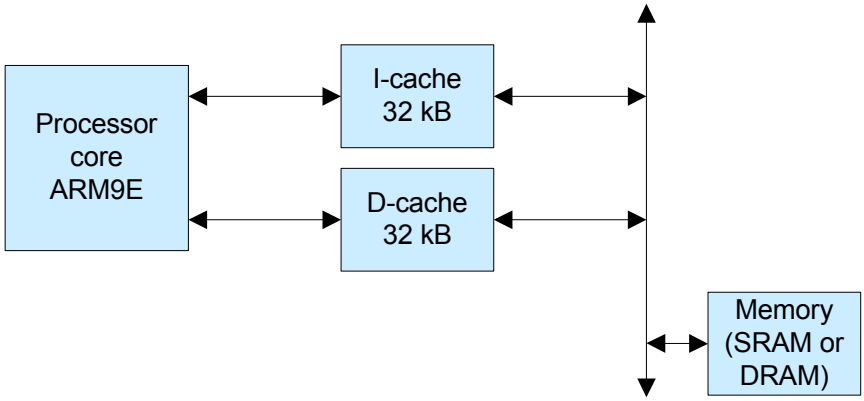



*Figure 38: Architecture of the evaluated test system*

The DRAM16 model is a medium fast memory model, which in comparison to the SDRAM model has a higher number of wait states for non-sequential accesses. These wait states reflect the time required for e.g. precharge and row activation if a non sequential access occurs. Additionally one wait state has been defined for sequential read. Although DRAM in general do not produce wait state in sequential access, this wait state has been defined for reflecting the influence of the delay time (time between applying the address and the data to be valid) during sequential accesses. The memory model of the ARMulator only supports wait states, but no delay times. Therefore the influence of delay time is approximated by setting the wait states to one.

*Table 21: Processor specification*

| **Processor type** | ARM946E-S |
|---|---|
| **Core frequency** | 250 MHz |
| **Bus frequency** | 125 MHz (½ core frequency) |
| **I-cache** | 32 kB, 4-way set-associative |
| **D-cache** | 32 kB, 4-way set-associative, write-back, write buffer |
| **MMU/PU** | Memory management/protection unit |

The DRAM24 model is the slowest memory model. The number of wait states for non-sequential accesses is higher as compared to DRAM16. However, in the model DRAM24 the approximation of the delay time for sequential accesses is not applied. Thus the influence of the different memory modeling can also be determined. An example of the influence can be seen in results presented below (Figure 41), where the DRAM24 in some cases is faster than the DRAM16.

To achieve a broad range of profiling results for the given H.264/AVC profile and level combination a set of different H.264/AVC test streams is generated using H.264/AVC video encoding software. The used encoder features a rate control mechanism and is using all tools for the targeted profile and level combination. Due to the different coding characteristics of video

sequences, five different sequences have been chosen (namely “Coastguard”, “Container”, “Stefan”, “StockholmPan”, “Tempete”). The sequences have a length of 125 to 150 frames with an I-framerate of 15 and are encoded with 352x288 pixels at a framerate of 15 frames per second. Each sequence is encoded with 256 and 384 kb/s, in order to analyze the influence of the compression rate.

*Table 22: Timing specification of the different memory architectures*

| Size | SRAM | DRAM16 | DRAM24 |
|---|---|---|---|
| **Non-sequential read time** | 8 ns (0 WS[1]) | 136 ns (16 WS[1]) | 208 ns (25 WS[1]) |
| **Sequential read time** | 8 ns (0 WS[1]) | 16 ns (1 WS[1]) | 8 ns (0 WS[1]) |
| **Non-sequential write time** | 8 ns (0 WS[1]) | 120 ns (14 WS[1]) | 200 ns (24 WS[1]) |
| **Sequential write time** | 8 ns (0 WS[1]) | 8 ns (0 WS[1]) | 8 ns (0 WS[1]) |

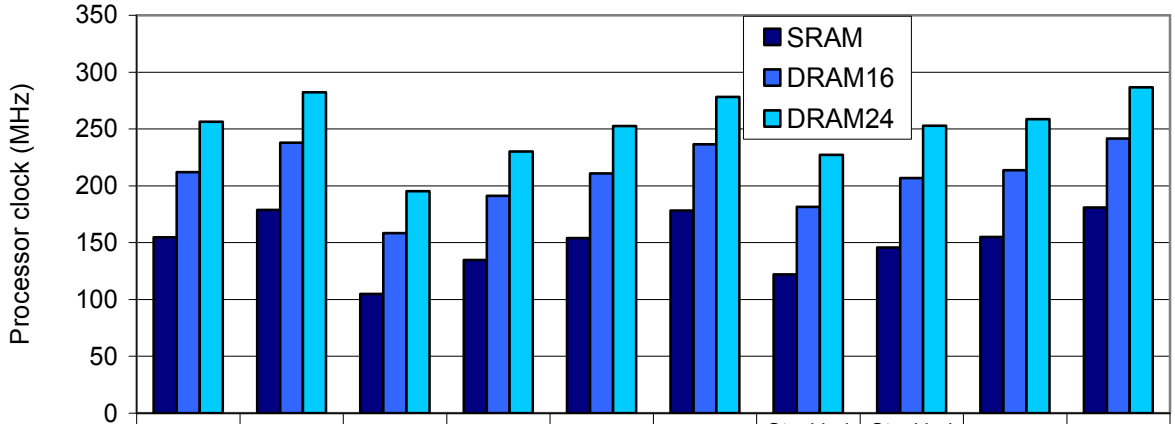
[1] Number of wait states in terms of memory bus cycles (bus frequency = ½ processor frequency)

## 6.1.2 Profiling Results

The generated profiling data is analyzed in three different ways, for entire bitstreams, for each function group and for each processed frame.

### 6.1.2.1 Overall Analysis

The overall analysis given in Figure 39 shows the averaged processor clock frequency needed for real time decoding of the corresponding bitstream. The processor clock frequency is derived from the number of external bus cycles (EBC). The EBC is ½ of the core clock frequency needed for the decoding of a complete sequence divided by the number of decoded pictures.




| | coastguard_256kb | coastguard_384kb | container_256kb | container_384kb | stefan_256kb | stefan_384kb | StockholmPan_256kb | StockholmPan_384kb | tempete_256kb | tempete_384kb |
|---|---|---|---|---|---|---|---|---|---|---|
| SRAM | 154,7086 | 178,9159 | 104,8536 | 134,7393 | 154,2127 | 178,2985 | 122,2628 | 145,8359 | 155,0371 | 180,7786 |
| DRAM16 | 212,0183 | 237,8373 | 158,5298 | 191,2608 | 210,7413 | 236,4282 | 181,5022 | 206,7864 | 213,8247 | 241,7735 |
| DRAM24 | 256,4842 | 282,1895 | 195,1987 | 230,2991 | 252,644 | 278,1794 | 227,3193 | 253,0266 | 258,7276 | 286,5564 |

*Figure 39: Average core clock frequencies for ARM9 implementation for different sequences and different memory architectures*

### 6.1.2.2 Function Group Analysis for I- and P-Frames with Different Memory Types

Function group analysis is based on the grouping of all incorporated functions of the H.264/AVC decoder, as given in Table 23.

*Table 23: Functional grouping of the H.264/AVC decoder*

| Functional group | Description |
|---|---|
| **Itrans** | Inverse H.264/AVC transformation |
| **IntraPrediction** | H.264/AVC intra prediction algorithms |
| **LoopFilter** | Deblocking filter |
| **MotionCompensation** | Motion compensation for P-frames |
| **BitstreamProcessing** | Low-level bitstream access functions |
| **EntropyDecoding** | CAVLC decoding |
| **ParsingDecoding** | High level parsing and control of bitstreams |
| **Memory** | Memory transfer functions like memcpy, memset |
| **Misc** | Other control helper functions and testbench |

The results are generated for I- and P-frames. In the following diagram the comparison of the required external bus cycles for the different function groups and the used memory architecture according to their picture coding type is depicted. The data is generated using the worst case pictures of the corresponding bitstream. Execution time is measured in number of external bus cycles (EBC). Figure 40 shows the results for the "Coastguard 256kb" sequence.

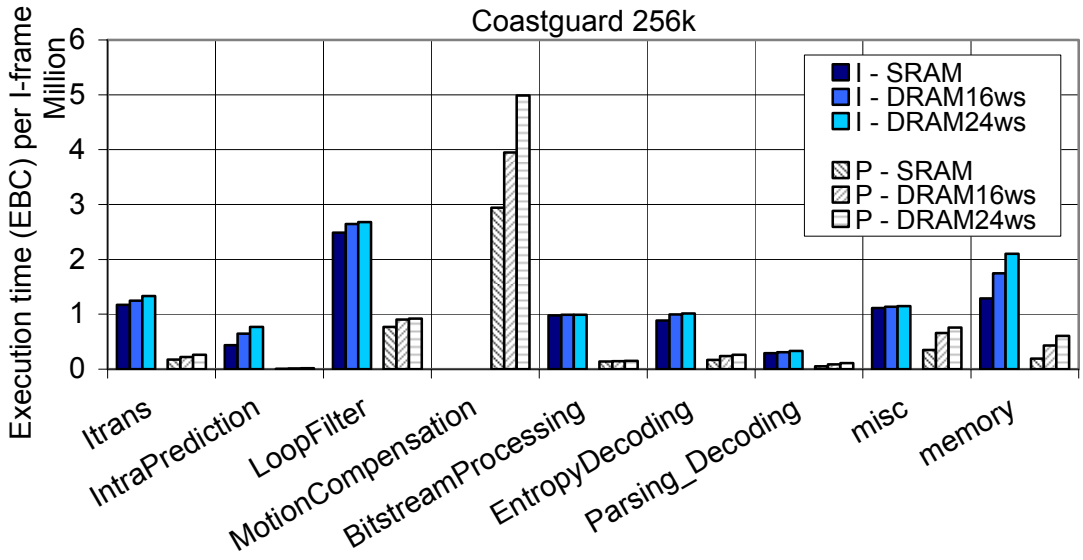



*Figure 40: Function group analysis for worst case I-frame (I) and P-frame (P) with different memory types for sequence "Coastguard 256kb"*

As can be seen, a big difference exists between the different frame types (I- and P-frames) and the different memory types. The memory type has a strong influence in the MotionCompensation and the Memory group. Both groups are highly memory access dominated, and in the MotionCompensation functions these accesses are random. Therefore the caches can not work

efficiently. Contrary to the LoopFilter group, which is also very memory-intensive, however as the filtering is applied to adjacent pixels, these accesses benefit from cache.

#### 6.1.2.3 Cycles per Frame Analysis

The following analysis is generated to analyze the number of used external bus cycles needed for each frame of a sequence of coded pictures. All three memory architectures are used for the profiling runs.

The number of external bus cycles is generated as a sum of all function groups of the decoder and reflects the overall needed real time performance for the according picture number. Execution time is measured in number of external bus cycles. Due to the different memory configurations concerning the parameters for sequential reads for DRAM16 and DRAM24 memory architectures in some cases the peak performance for DRAM24 can be higher than for DRAM16. This is the case if the number of sequential read accesses dominates the overall number of accesses, since the DRAM16 is slower in sequential read accesses than the DRAM24. That special case can be observed in Figure 41. The decoding of the frames (I-frames), where the DRAM16 is slower than the DRAM24 configuration, includes mostly sequential reads accesses.

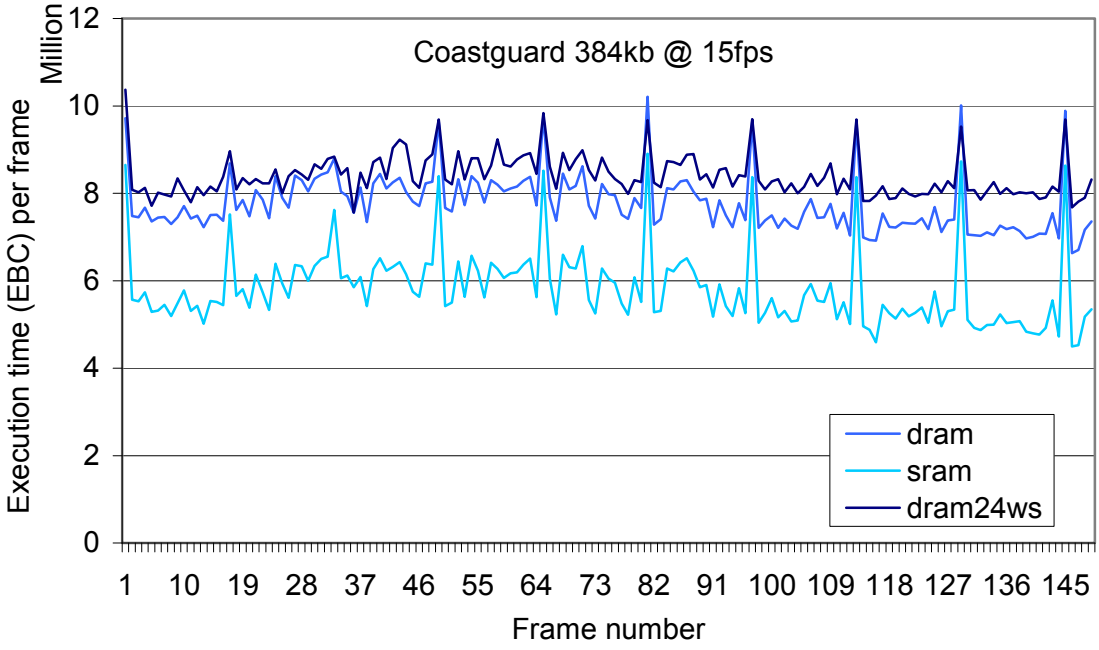



*Figure 41: Cycles per frame analysis for different memory types ("Coastguard 384kb")*

#### 6.1.2.4 Memory Access Statistics

The following analysis is generated for analyzing the number of data memory accesses and data cache read misses. Separate results are created for different H.264/AVC decoder memory sections. As shown in Table 24, the decoder memory is split up in the read-only section (ER_RO), read-write section (ER_RW), zero-initialized section (ER_ZI), the stack and the heap variables. The results of this analysis allow an estimation of which data sections uses the data cache efficiently.

*Table 24: H.264/AVC decoder memory sections and heap variables*

| Memory section | Description | Size |
|---|---|---|
| **ER_RO** | Program code and constant global variables | 103460 bytes |
| **ER_ZI** | Global variables which are initialized to zero | 612 bytes |
| **ER_WR** | All other global variables (not constant and not initialized to zero) | 126920 bytes |
| **Stack0** | Stack | 8192 bytes |
| **Dec_struct** | Main memory structure for the decoder (including the current frame, the five reference frames and additional decoding data) | 967796 bytes |
| **AUBuf** | Buffer for the current access unit | 16384 bytes |
| **Bs_struct** | Structure containing status information about input bitstream buffer | 36 bytes |
| **Others** | Mainly a 4 kB input bitstream buffer (bs->pi32_buffer) | ~ 4096 bytes |
| **Sum** | | ~ 1.17 MB |

This analysis distinguishes between memory access and cache misses, as illustrated in Figure 42. Memory accesses are caused by load and store operations executed by the processor. If a data value is not available in the cache a load or store operation leads to a cache miss. In case of a read operation a cache miss leads to a cache fill of a cache line. In case of a store operation a cache miss leads to write to the memory via the write buffer.

In the following, memory accesses are counted for both, load and store operations, whereas cache misses are only counted for read operations (cache fills). This restriction is caused by the profiling tool.

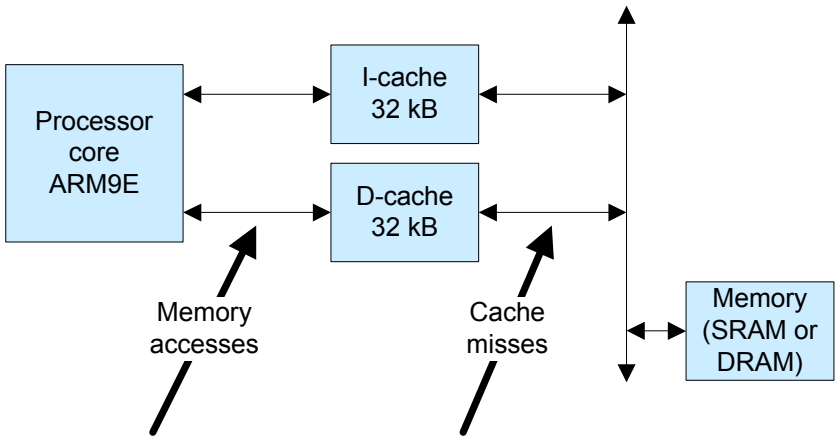



*Figure 42: Locating data transfers caused by memory accesses and cache misses*

Figure 43 shows the memory accesses to the sections and heap variables for decoding the sequence “Stefan 384kb”. Memory accesses are the number of read or write operations caused by load or store operations. Regardless, if the load or store operation is a byte, half-word or word access, each one of them is counted as one access. The cache misses, which are caused by load operations to the specific memory sections and heap variables, correspond to the right ordinate in Figure 43.

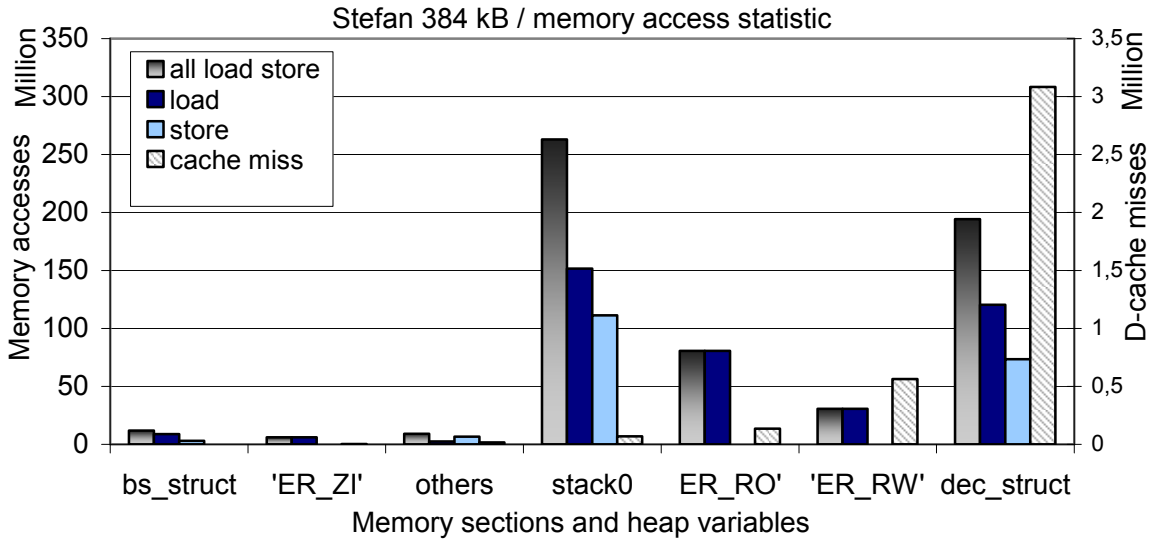



*Figure 43: Memory accesses and D-cache misses for the sections and heap variables for decoding the sequence "Stefan 384kb"*

As can be seen, most load operations access the stack. However, these load operations only cause a few read cache misses. This shows that the stack uses the cache very efficiently.

Further the diagrams show that for reading the dec_struct the cache can not be used as efficiently as for reading the stack. The reason for this is that the dec_struct is large and the locations (addresses) of accesses to the dec_struct are very random. However, comparing the total number of accesses to the dec_struct (200 million) with the number of read cache misses (3 million) shows that using the cache still has a significant positive influence on access time to the dec_struct. Therefore the dec_struct should not be marked as non-cacheable.

In the following an overview of the accesses to the main memory (data transfer between caches and DRAM/SRAM) is given. Read accesses to the main memory are caused by read data cache misses and instruction cache misses. Each cache miss leads to a data cache fill of a cache line. When decoding the "Stefan 384kb" sequence the memory accesses provided in Table 25 occur.

*Table 25: Accesses to the main memory*

| **Read access type** | **Accesses** | **Sum** |
|---|---|---|
| **Instruction cache misses** | 1548764 | (1548764*8) + (3887992*8) = 43 494 048 |
| **Data cache read misses** | 3887992 | |
| **Write access type** | **Accesses** | **Sum** |
| **Non-sequential write accesses** | 23695488 | 23695488+11168236 = 34863724 |
| **Sequential write accesses** | 11168236 | |

For each of this cache misses a cache line with a length of eight words (4 bytes each) is read from the main memory, this leads to about 43 million read accesses to the main memory. The write accesses to the main memory are calculated from the results of the memory model simulator. The simulator provides the number of sequential and non-sequential accesses to the memory. This leads to a total number of about 34 million write accesses to the main memory.

### 6.1.3 Profiling-Based Software Optimization Potential

#### 6.1.3.1 Algorithmic Optimizations

Assembly coding and the usage of SIMD instructions are optimization techniques that can be considered to increase the decoder's performance. Code analysis shows that in most cases the resulting assembly code generated by the compiler is nearly optimal if the C source code is written according to the recommendation given by ARM. Besides the usage of hand optimized code using SIMD-instructions there is no significant performance gain to expect. If the amount of work is taken into account and the fact the maintainability of the code is getting worse it is not recommended to use assemble code for the optimization of the research H.264/AVC decoder.

An estimation of optimization gains due to the usage of the SIMD-instructions of the ARM11 architecture is performed. Due to the fact that an ARM11 profiler is not available, the achieved performance gain by using the SIMD instructions is estimated. The approximately used instructions for an unoptimized function are counted and compared those numbers to the numbers expected by using SIMD instructions. In the following the resulting optimization factor are given for each function. The optimization factors are calculated as:

$$optimizationFactor = \frac{executionTime(unoptimizedCode)}{executionTime(optimizedCode)} \tag{29}$$

where the execution times are estimated values due to the number of instructions.

In Figure 44 the expected optimization gain in terms of real time performance is depicted using all before mentioned optimizations. The estimated performance gains, as given in Section 8.1.4, are included in the simulation result, as described in Section 4.3.4. In detail, the cycle count of each function in each frame is multiplied by the performance gain factor. Then the overall results are recalculated. The new results (with performance gain) are compared to the old results (without the performance gain) with Equation 30:

$$optimizationGain = 100 \cdot \frac{executionTime(unoptimizedCode) - executionTime(optimizedCode)}{executionTime(unoptimizedCode)} \tag{30}$$

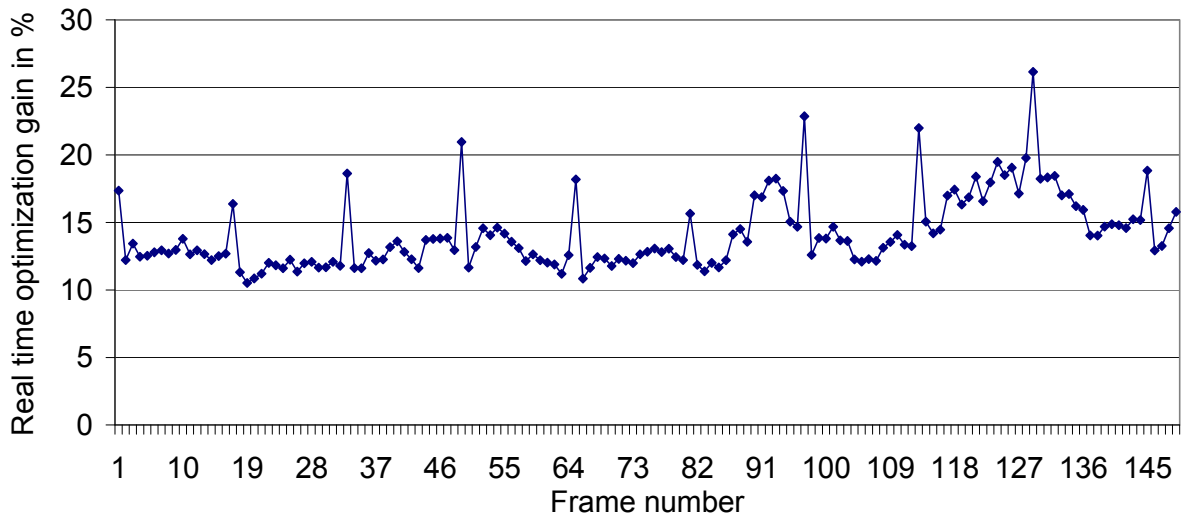



*Figure 44: Optimization gain per frame using proposed ARM11 SIMD optimizations for sequence "Stefan 384kb"*

In most cases of the optimized functions the SIMD implementation of the add instruction e.g. add8to16 can be used. Due to the large overhead in terms of load/store operations and reformatting operations the usage of other ARM11 SIMD instructions is very limited. Those SIMD instructions require lots of data access instructions and other instructions in order to organize the data in a manner that the special SIMD instructions can be used. In most cases the overhead of needed instructions to organize the data into the corresponding registers costs more instructions cycles than the ones saved due to the SIMD usage.

The usage of the MAC instruction is limited to very few cases, since in most functions the basic operation consists of add instructions followed by shift instructions.

### 6.1.3.2 Cache Optimizations

An analysis of the influence of cache size on performance is performed. For this purpose the "Stefan 384kb" sequence is decoded on the ARMulator for the ARM946E-S processor with different cache sizes, see Figure 45.

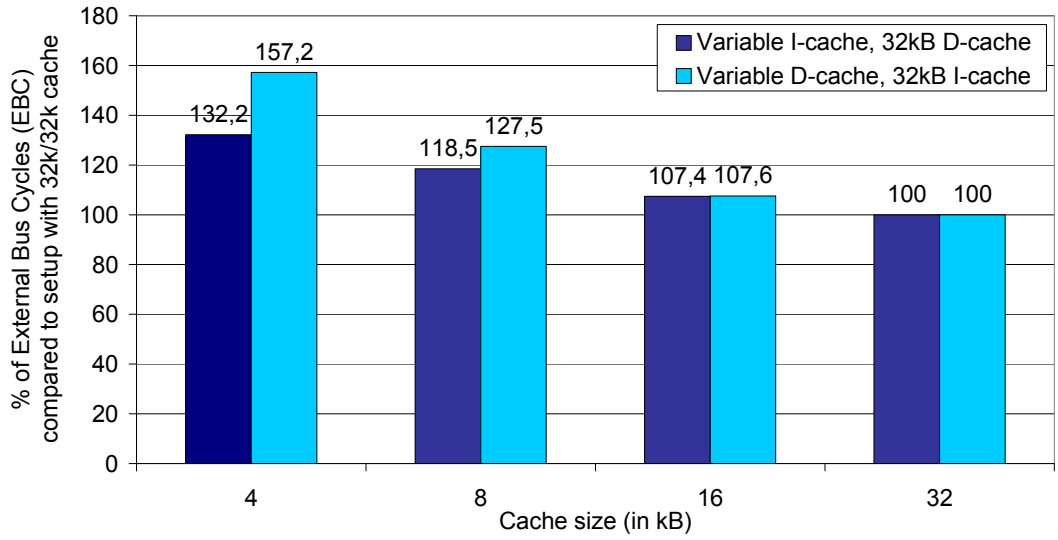



*Figure 45: Influence of I- and D-cache size on performance*

In the first test run the D-cache size is set to a fixed value of 32 kB, and the size of the I-cache is varied from 4 to 32 kB. In the second test run the I-cache size is set to a fixed value of 32 kB, and the D-cache size is varied. The results of the first (blue) and the second (red) test run are depicted in the figure. The figure shows the increase of external bus cycles in percent as compared to a system with 32 kB I-cache and 32 kB D-cache.

The result shows that if either the I-cache or the D-cache size is decreased from 32 kB to 16 kB this leads to the same increase of external bus cycles of approx. 7.5 %. However, if further decreased, the size of the D-cache has a larger impact on the performance than the I-cache size.

### 6.1.3.3 Speedup Estimation due to TCMs

A speedup is expected due to a reduction of the accesses to the external memory. This reduction should be achieved by adding a tightly coupled memory (TCM) to the system. The mapping of the data areas to the TCM and the slow external memory is performed as described in Section 4.4.3.

The profiling of the H.264/AVC decoder shows that the overall number of data cache misses is more than 2.5 times higher than the instruction cache misses, see Section 6.1.2.4. Therefore in the following only data access optimization is considered. The H.264/AVC decoder contains about 200 data areas (global and heap variables) which are potential candidates for the storage in the TCM. These data areas include e.g. reference frames, temporary decoding data (e.g. motion vectors) and constant lookup tables (e.g. for VLC). MEMTRACE is applied for tracing the memory accesses to each data area and the cache misses which occurred during read accesses to these data areas. This information is used for choosing the data areas to be stored in the TCM. The results for the "Stefan 384kb" sequence are partly shown in Table 26.

*Table 26: Decoder data areas sorted according to their number of cache misses*

| Data area | Size | All load store | Load | Store | Cache miss |
|---|---|---|---|---|---|
| **Dec_struct.Y_plane5** | 101376 | 16509308 | 11836357 | 4672951 | 332720 |
| **Dec_struct.Y_plane3** | 101376 | 14061611 | 10048336 | 4013275 | 313515 |
| **Dec_struct.Y_plane4** | 101376 | 14123674 | 10135951 | 3987723 | 311135 |
| **Dec_struct.Y_plane2** | 101376 | 12352958 | 8215325 | 4137633 | 281123 |
| **Dec_struct.Y_plane1** | 101376 | 9760766 | 6916857 | 2843909 | 211234 |
| **Dec_struct.Y_plane0** | 101376 | 9451125 | 6647755 | 2803370 | 206220 |
| **Dec_struct.mvd** | 25344 | 4861092 | 2732449 | 2128643 | 193300 |
| **Dec_struct.mb_data** | 12672 | 2820493 | 1925502 | 894991 | 165202 |
| **...** | ... | ... | ... | ... | ... |

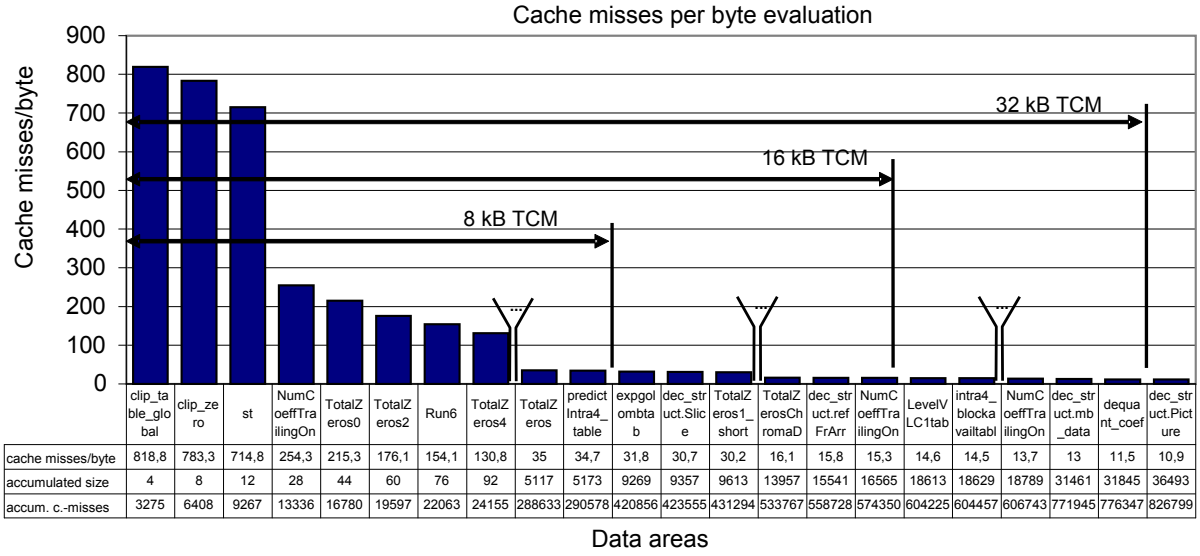
They are sorted by the overall number of cache misses they produce. The data area with the most cache misses (and therefore with most influence on performance) are the Y planes of the reference frame buffers. However, these are not potential candidates for the TCM, since they are to large (approx. 100 kB each) for the TCM. Therefore the data areas are resorted as shown in Table 27. The data areas are sorted by the cache misses per byte.

The table also shows the accumulated size and cache misses of the variables. The accumulated cache misses indicate the speedup if these variables were to be stored in the TCM. The accumulated size shows how much memory space in the TCM would be required for storing the specific data and all data areas above it.

*Table 27: Decoder data areas sorted according to their number of cache misses per byte*

| Data area | Size | Cache miss | Accum. size | Accum. cache miss | Cache miss / byte |
|---|---|---|---|---|---|
| **Clip_table_global** | 4 | 3275 | 4 | 3275 | 818.75 |
| **Clip_zero** | 4 | 3133 | 8 | 6408 | 783.25 |
| **St** | 4 | 2859 | 12 | 9267 | 714.75 |
| **NumCoeffTrailingOnes0** | 16 | 4069 | 28 | 13336 | 254.31 |
| **TotalZeros0** | 16 | 3444 | 44 | 16780 | 215.25 |
| **TotalZeros2** | 16 | 2817 | 60 | 19597 | 176.06 |
| **Run6** | 16 | 2466 | 76 | 22063 | 154.12 |
| **TotalZeros4** | 16 | 2092 | 92 | 24155 | 130.75 |
| **Scanpattern** | 64 | 8046 | 156 | 32201 | 125.71 |
| **Dec_struct.stream** | 36 | 4101 | 192 | 36302 | 113.91 |
| **NumCoeffTrailingOnes2** | 16 | 1808 | 208 | 38110 | 113.00 |
| **NumCoeffTrailingOnesChromaDC** | 16 | 1718 | 224 | 39828 | 107.37 |
| **TotalZeros1** | 16 | 1697 | 240 | 41525 | 106.06 |
| **Run2** | 16 | 1690 | 256 | 43215 | 105.62 |
| **Intra4_blockavailtable00** | 16 | 1688 | 272 | 44903 | 105.50 |
| **Ijpos** | 16 | 1664 | 288 | 46567 | 104.00 |
| **NumCoeffTrailingOnes1** | 16 | 1640 | 304 | 48207 | 102.50 |
| **Run1** | 16 | 1546 | 320 | 49753 | 96.62 |
| **...** | ... | ... | ... | ... | ... |

Figure 46 shows how this information can be used for choosing the optimal candidates for a TCM of a specific size. The left most data area is the one with the highest cache miss density (cache miss per byte). Therefore this data area has the highest priority to be stored in the TCM. For choosing all data areas to be stored in a TCM with a specific size, one can continue to the right until the accumulated size reaches the TCM size.




| | clip_table_global | clip_zero | st | NumCoeffTrailingOn | TotalZeros0 | TotalZeros2 | Run6 | TotalZeros4 | TotalZeros | predictIntra4_table | expgolombtab | dec_struct.Slice | TotalZeros1_short | TotalZerosChromaD | dec_struct.refFrArr | NumCoeffTrailingOn | LevelVLC1tab | intra4_blockavailtabl | NumCoeffTrailingOn | dec_struct.mb_data | dequant_coef | dec_struct.Picture |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cache misses/byte | 818,8 | 783,3 | 714,8 | 254,3 | 215,3 | 176,1 | 154,1 | 130,8 | 35 | 34,7 | 31,8 | 30,7 | 30,2 | 16,1 | 15,8 | 15,3 | 14,6 | 14,5 | 13,7 | 13 | 11,5 | 10,9 |
| accumulated size | 4 | 8 | 12 | 28 | 44 | 60 | 76 | 92 | 5117 | 5173 | 9269 | 9357 | 9613 | 13957 | 15541 | 16565 | 18613 | 18629 | 18789 | 31461 | 31845 | 36493 |
| accum. c.-misses | 3275 | 6408 | 9267 | 13336 | 16780 | 19597 | 22063 | 24155 | 288633 | 290578 | 420856 | 423555 | 431294 | 533767 | 558728 | 574350 | 604225 | 604457 | 606743 | 771945 | 776347 | 826799 |

*Figure 46: Data areas sorted by cache misses per byte and indication of chosen data areas for specific TCM sizes*

Table 28 shows the result if a TCM with the size of 4, 8, 16 and 32 kB would be filled optimally with the data areas. For an ARM946E-S processor with 32 kB of data TCM a simulation is performed in order to achieve an estimation for the speedup, which can be expected. The simulation with the memory DRAM24 shows that an overall speedup of 5 % could be achieved.

*Table 28: Cache miss reduction due to data TCMs with the size of 4, 8, 16 and 32 kB*

| TCM size | Cache miss reduction | Cache miss reduction in % ( 3834465 misses overall) | Estimated speedup |
|---|---|---|---|
| **4 kB** | 249233 | 6.5 % | 2.4 % |
| **8 kB** | 381611 | 10 % | 3.7 % |
| **16 kB** | 558728 | 14.6 % | 5.4 % |
| **32 kB** | 776347 | 20.2 % | 7.5 % |

The speedup is expected to be dependent on the processor architecture. Therefore, an upper limit for a potential speedup is estimated, which might be possible with the ARM11 architecture. An upper limit for the expected speedup due to TMCs is calculated based on the cache miss reduction. As shown in Table 28, due to 32 kB of data TCM the number of data read cache misses is reduced by 20 %. This 20 % cache miss reduction could lead, in an optimal system architecture, to a reduction of 20 % of the time spent on memory accesses. Since the simulation showed us that about 50 % of the execution time is spent on memory accesses, this could lead to a 10 % speedup of overall execution time in the best case.

This leads to the assumption that the potential speedup due to 32 kB of data TCM with the ARM11 architecture will be between 5 % and 10 %. In the following a mean value of 7.5 % is assumed. Starting with this mean value and the assumption of a linear relationship between cache miss reduction and speedup due to TCM, the speedup for the other TCM sizes is calculated as follows:

$$speedup(x\ kByte\ TCM) = speedup(32\ kByte\ TCM) * \frac{cacheMiss\ \mathrm{Re}\ duction(x\ kByte\ TCM)}{cacheMiss\ \mathrm{Re}\ duction(32\ kByte\ TCM)} \quad (31)$$

This leads to the estimated speedups given in Table 28.

#### 6.1.3.4 Using DMA

The ARM11 processor family supports TCMs with direct memory access (DMA). DMA strategies can be applied for an efficient dynamic usage of the TCM. Thus data, which is accessed frequently only at a specific point of time, can be stored in the TCM only during this time. After the processing the data can be written back to the DRAM and other data can be stored in the TCM.

This dynamic usage of the TCM is examined on a frame basis. If I-frames and P-frames have a different cache miss statistics, different data could be stored in the TCM during an I-frame and a P-frame. This would lead to an optimized TCM usage and an increased performance. Therefore the cache miss statistics is analyzed for I-frames and P-frames separately and extracted the variables to be stored in TCM for both frame-types. The lists of variables is compared to the previous list of variables for all frames given in Table 27. This comparison shows that the optimal TCM memory map for I-frames differs from the one for all frames in a few variables, as given in Table 29. For the P-frames, as expected, the optimal memory map is same as the one for all frames. The term “all frames” determines the entire sequence of I- and P-frames in the H.264/AVC sequence.

*Table 29: Difference of optimal TCM memory map for I-frames and for all frames*

| Variable name | Size | Cache miss | Accumulated size | Accumulated cache miss | Cache miss per byte |
|---|---|---|---|---|---|
| **Intra4_blockavailtable01** | 16 | 16 | 16 | 16 | 1 |
| **NumCoeffTrailingOnes2_short** | 1024 | 2711 | 1040 | 2727 | 2.65 |
| **TotalZeros10** | 16 | 37 | 1056 | 2764 | 2.31 |
| **TotalZeros5_short** | 256 | 302 | 1312 | 3066 | 1.18 |
| **TotalZeros6_short** | 256 | 258 | 1568 | 3324 | 1.00 |
| **TotalZeros9** | 16 | 25 | 1584 | 3349 | 1.56 |

As can be seen, the optimal memory map differs in about 1.5 kB of TCM memory. It shows that the number of overall cache misses with an optimal TCM mapping for I-frames could be reduced by 3349. Table 30 shows the influence on the cache miss count for I-frames.

*Table 30: Influence of optimal TCM memory map for I-frames*

| | Cache misses | Reduction in % |
|---|---|---|
| **Overall number of cache misses in I-frames** | 231483 | |
| **Cache miss reduction with I-frame TCM map** | 57407 | 25.8 % |
| **Cache miss reduction with all-frame TCM map** | 54058 | 23.4 % |
| **Difference between I-frame and all-frame TCM map** | 3349 | 1.4 % |

This shows that using an optimized TCM map for I-frames would decrease the number of cache misses in I-frames only by 1.5 %. This in turn would only lead to minimal performance

increase. As a consequence it can be stated that DMA strategies on a frame-basis would not lead to a performance gain. However, more fine-grain DMA strategies, e.g. on a macroblock-basis for storing adjacent macroblocks of the current frame in the TCM, would required an immense software recoding.

### 6.1.4 Summary of Profiling and Software Implementation Results

This section summarizes the performance analysis and the results of the different optimization strategies applied. Additionally, the memory map for the usage of a data TCM is shown.

#### 6.1.4.1 Performance Estimation

Table 31 shows the performance gain, which can be expected with the different optimizations. These performance gains are compared to the system with external memory DRAM24 and 32 kB of I-cache and D-cache but without TCM. The table shows that the usage of SIMD instructions has a significant influence on the performance whereas DMA strategies (applied on a frame-basis) have only a marginal influence.

*Table 31: Average performance gain due to optimizations*

| Optimization | Average performance gain |
|---|---|
| **SIMD instructions** | +14 % |
| **I-cache 32 kB / 16 kB / 8 kB / 4 kB** | 0 % / -7 % / -19 % / -32 % |
| **D-cache 32 kB / 16 kB / 8 kB / 4 kB** | 0 % / -8 % / -28 % / -57 % |
| **D-TCM 32 kB / 16 kB / 8 kB / 4 kB** | +7.5 % / +5.4 % / +3.7 % / 2.4 % |
| **DMA** | < 1 % |

In Figure 47 the influence of the SIMD and TCM optimizations on the required processor frequency is shown for the “Stefan 384kb” sequence. The results for the reference system (32 kB I-/D-cache + DRAM24) are based on the results presented in Figure 41. The required average and peak processor frequency is given in Figure 48.

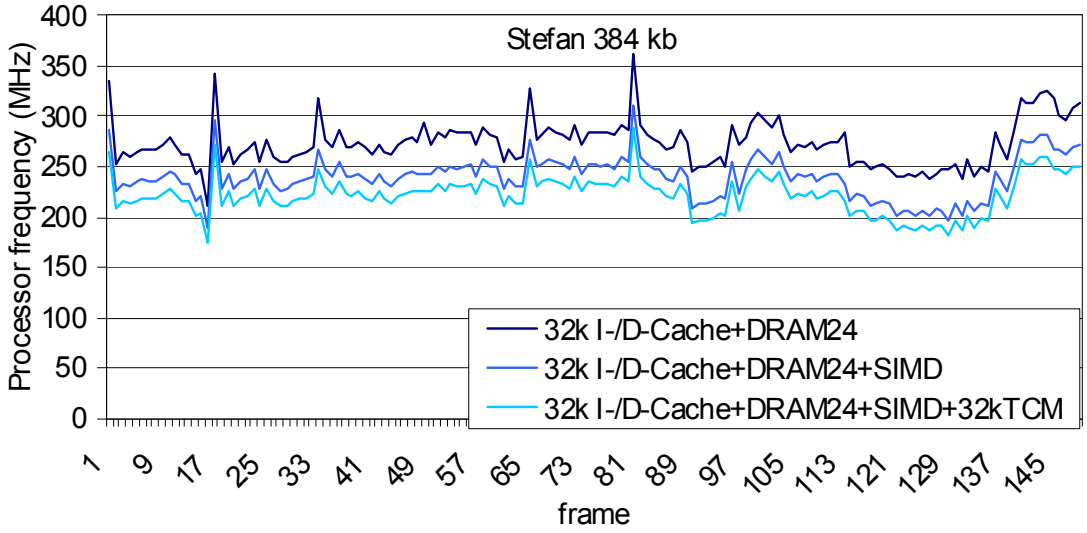



*Figure 47: Performance comparison for different system configurations*

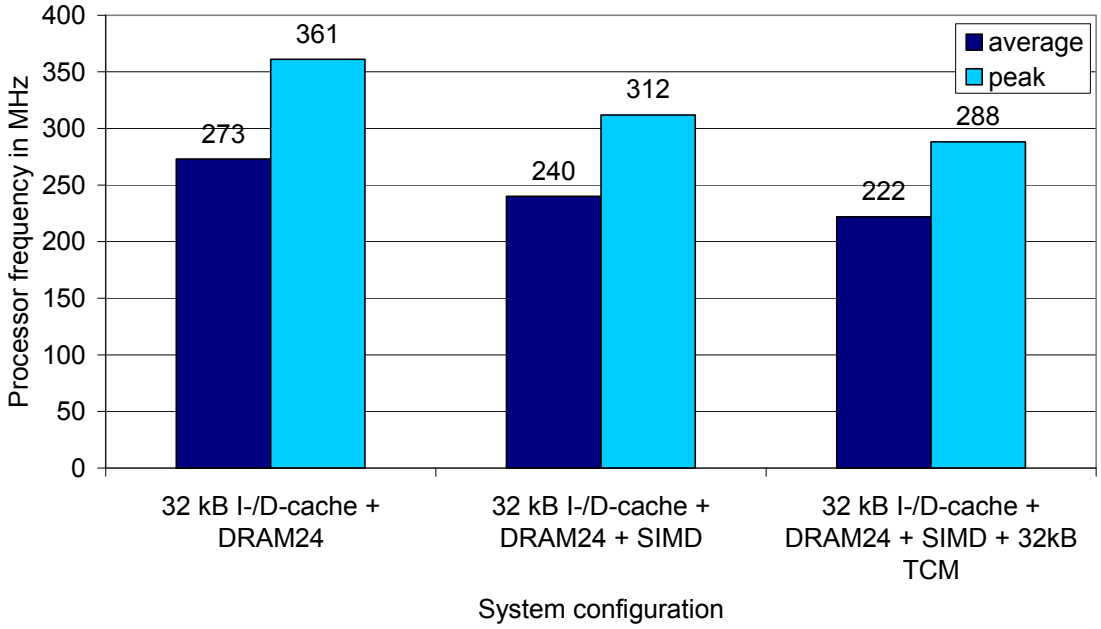



*Figure 48: Comparison of the required average and peak processor frequency for different system configurations*

### 6.1.4.2 Memory Requirements

The overall memory requirements for the H.264/AVC decoder are approximately 1.2 MB, consisting of about 1.1 MB of data memory and 90 kB of program code. For a system configuration with a 32 kB TCM a memory map as shown in Table 32 would be applied.

*Table 32: Memory map for a system configurations with 32 kB of TCM*

| Section | DRAM | TCM |
|---|---|---|
| **Boot code / stdIO-Lib / testappl.** | 224+13404+3348=16976 bytes | |
| **H.264/AVC decoder library code** | 73344 bytes | |
| **Global variables** | 134988 bytes | 12092 bytes |
| **Decode structure (heap)** | 947232 bytes | 20564 bytes |
| **AUBuffer (heap)** | ~ 16384 bytes | |
| **Heap (without decode structure and AUBuffer)** | 4096 bytes | 36 bytes |
| **Stack** | 8192 bytes | |
| **Sum** | 1201212 bytes ≈ 1.15 MB | 32692 bytes = 32 kB |

### 6.1.4.3 Conclusion

Taking into account that these values rely on simulations on an ARM946E-S processor and estimations, the results can be summarized as follows. Using ARMv6 SIMD and memory

footprint optimizations the H.264/AVC decoder will be capable of real time decoding on a processor system with approx.

- 200-250 Mhz core clock frequency (average performance)
- 250-300 Mhz core clock frequency (peak performance)
- external bus clock with ½ core clock
- 32 kB I-cache and 32 kB D-cache32 kB TCM
- 1.2 MB external DRAM

A meaningful system variation can be the reduction of the I-cache size in order to reduce the die area. However, a reduced I-cache size of either 16 kB or 8 kB would increase the required processor frequency by approx. 8 % or 18 %.

### 6.1.5 Hardware/Software System Architecture

Considering the dynamic power consumption of CMOS-circuits, given in Equation 32, the high system frequency leads to high power consumption.

$$P_{dynamic} = C_k \cdot f_k \cdot V_{DD}^2 \tag{32}$$

For achieving lower power consumption, methods need to be applied, which allow the reduction of the system frequency, which in turn also allows a lower supply voltage (voltage scaling). Hardware accelerators can be used for this purpose. However, their influence on the capacitance has to be considered and reduced by mechanism like clock gating. Furthermore the memory architecture needs to be adapted (reduced) to the specific application requirements.

The profiling results presented in the previous sections show that a few hot spots can be identified in the software. Considering the results presented in Section 6.1.2.2 the following hot spots can be identified:

- motion compensation
- deblocking (loop-) filter
- memory transfers
- integer transform (itrans)

These functional units are candidates for hardware acceleration, which leads to the system architecture as depicted in Figure 49.

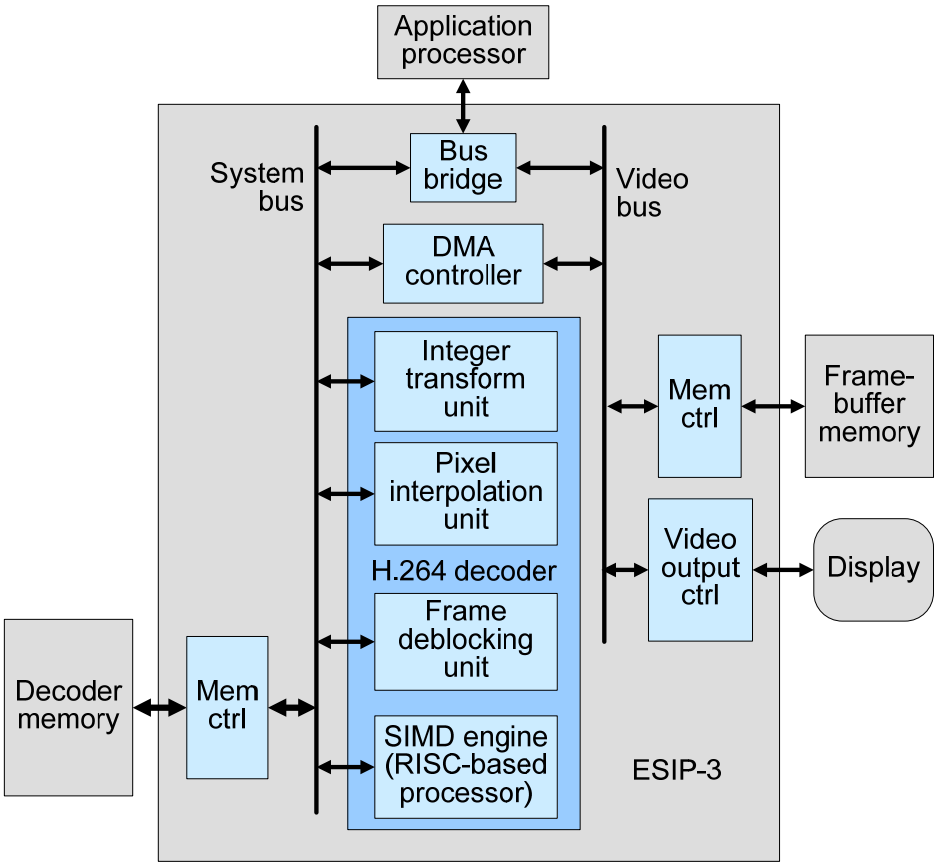



*Figure 49: System layout of the H.264/AVC decoder chip based on the profiling results with a system bus and a separate video bus*

The application processor running the software parts is extended with a companion chip for acceleration of the video decoding. The companion chip [89] contains the above-mentioned hardware accelerators: for H.264/AVC decoding. Table 33 shows a comparison of the required cycle times of the accelerators with their software counterparts.

The coprocessors use the interfaces described in Section 5.6.1, including the memory-mapped status and control registers and the input and output memory areas. The cosimulation of the processors and the coprocessor is performed as described in Section 4.2.3. Furthermore a so-called SIMD engine is available on the chip, which is a 32-bit RISC processor enhanced with special SIMD instructions. The 32-bit system bus connecting the processor core with the main memory and coprocessor components is augmented with a DMA-controller which supports the main processor by performing the memory transfers to the coprocessor units. A video output unit is implemented directly driving a connected display or video DAC. To avoid a heavy bus load on the mentioned system bus due to transfers from a frame buffer to the video output interface, an extra frame buffer memory and the video output unit are provided by a separate video bus system. The data transfers between these bus systems are also performed by the DMA controller. The main control functionality of the decoder can either be run on the application processor or on the RISC core on the companion chip.

*Table 33: Comparison of the Execution time in hardware and software*

| Implementation | Deblocking | Pixel interpolation | Inverse transform |
|---|---|---|---|
| **Software** | 3000-7000 cycles | 100-700 cycles | 320 cycles |
| **Hardware** | 232 cycles | 16-34 cycles | 30 cycles |

To fully evaluate the proposed concept the complete SoC architecture the Embedded System Group at the Fraunhofer Heinrich-Hertz-Institut (HHI) developed and implemented an ASIC design using UMC's L180 1P6M GII logic technology, see Figure 50. The maximum clock frequency of the design is 120 MHz, whereas 50 MHz should be sufficient for the DVB-H scenario. The evaluation board for the chip allows the fully functional verification and furthermore exhaustive performance testing and power measurements, separately for memory, core and IO supply voltages.

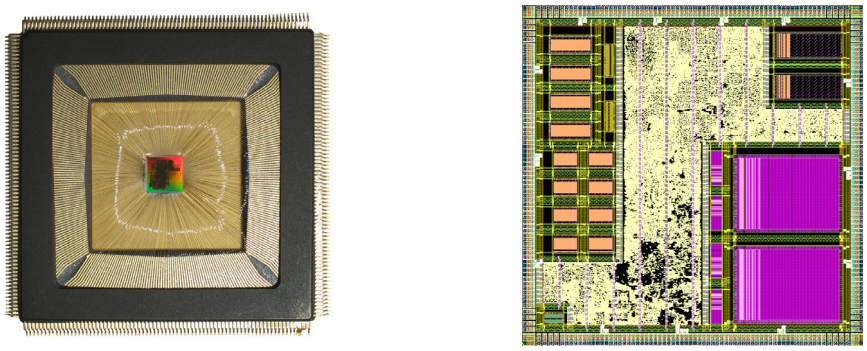


*Figure 50: An H.264/AVC decoder companion chip based on the profiling results (die and chip layout)*

## 6.2 GestAvatar – Gesture Detection for Avatar Control

Within another project a gesture recognition and head tracking system [82] is profiled. The system should be run on a PDA with the following specification:

- ARM9 processor (ARM946E-S)
- processor speed: 250 MHz
- caches: 32 kB of instruction cache and 32 kB of data cache
- memory bus running with ½ processor speed
- external SDRAM (25 wait states for non-sequential accesses)

The software contains three major components of the application are:

- Hand head tracking
- Facial feature tracking
- Gesture recognition

The goal of the profiling is an estimation of the real time requirements of the application. Figure 51 shows the distribution of the clock cycle requirements over the different components and Figure 52 the instruction cycle time requirement per frame.

### 6.2.1 Results

The processing is highly influenced by the floating point convolution required in the feature-tracker unit. Horizontal and vertical convolution require about 60 % of the overall processing time. Although the profiling results on the PC also identify the feature tracker as the most demanding part of the software, the results are not as drastically. This is due to the fact that the PDA does not provide any hardware acceleration for floating point arithmetic, which is heavily used within the convolution. Therefore either emulation or a library implementation is used for floating point operations. Within this profiling the faster choice, a library implementation is used. Floating point emulation may decrease the performance significantly. If the floating-point arithmetic could be replaced by integer arithmetic, the required processing time for these operations may decrease by a factor of 5.

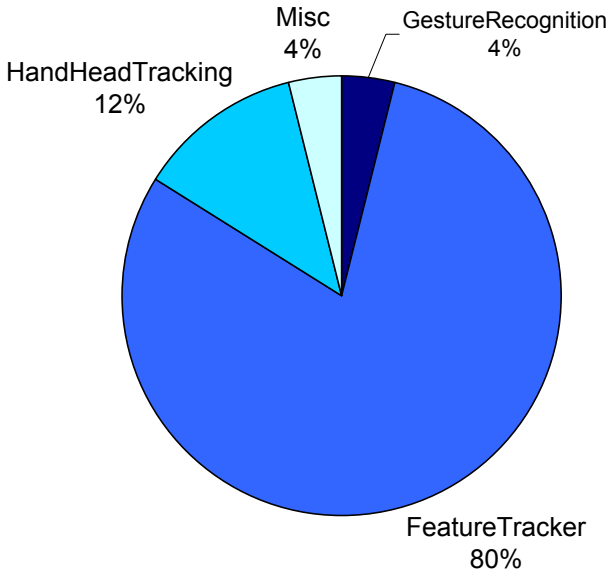



*Figure 51: Overall distribution of the clock cycle requirement per functional block (resolution: 320x240 pixels, sub-sampling: 2)*

Current PDAs have a higher processor speed (up to 624 MHz) than the one used in this profiling, therefore the performance results on real hardware may be a superior. However, since the simulated SDRAM access times, which highly influence the performance, correspond well to today's devices, only a speedup factor of up to 1.5 is expected.

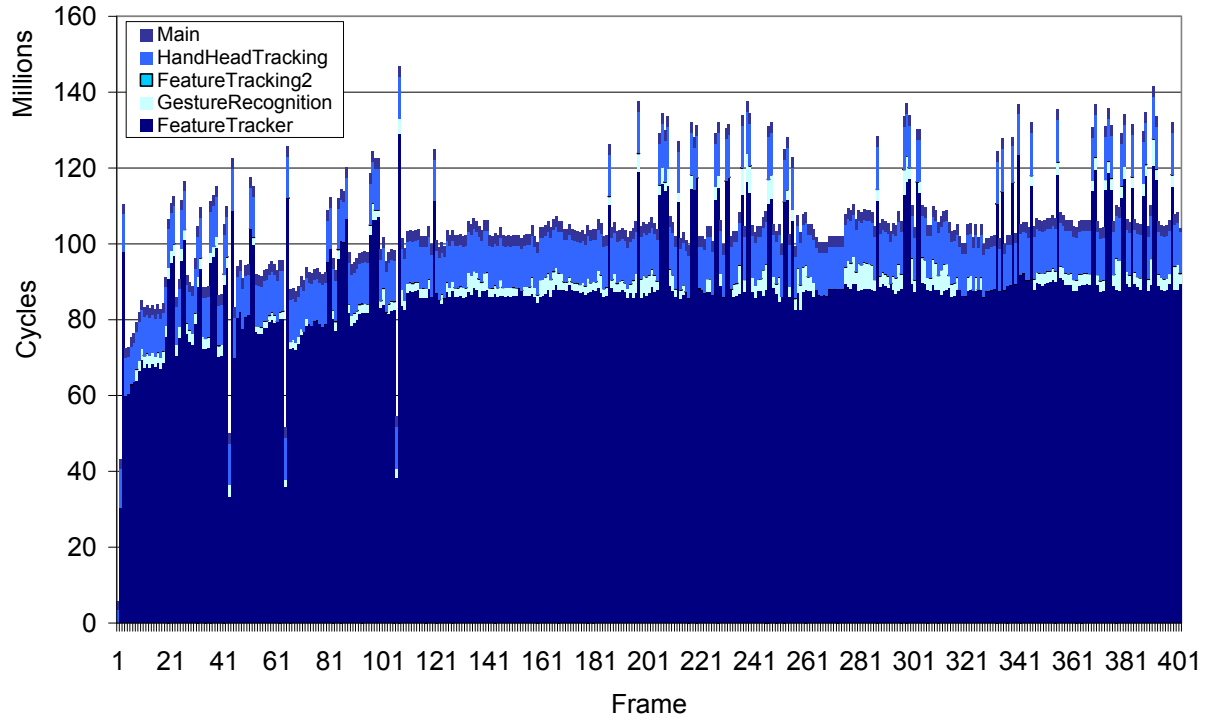



*Figure 52: Per frame analysis of the clock cycle requirement per functional block (resolution: 320x240 pixels, sub-sampling: 2)*

A solution for reducing the cycle requirement on the PDA is to outsource the FeatureTracker component to an external server, e.g. a PC. The video data should be transmitted as H.264/AVC encoded video. Table 34 and Table 35 show the resulting overall performance, comparing three different scenarios:

- full processing on the PDA
- shared processing on PDA and server with H.264/AVC encoded video transfer of the region of interest
- full processing on the server with H.264/AVC encoded video transfer of the entire image

*Table 34: Processing times of the most demanding functional blocks for different client/server setups*

| | Processing time for full PDA processing | Processing time for shared processing | Processing time for external processing |
|---|---|---|---|
| **Hand head tracking** | | | |
| **640x480** | 111 ms | | |
| **320x240, subsampling=2** | 34 ms | | |
| **320x240, subsampling=4** | 31 ms | | |
| **Facial feature tracking** | | H.264/AVC intra-only encoding of head-box | H.264/AVC intra-only encoding of full image |
| **640x480** | 915 ms | 114 ms (1400 kb/s) | 816 ms (8000 kb/s) |
| **320x240, subsampling=2** | 232 ms | 33 ms (350 kb/s) | 204 ms (2000 kb/s) |
| **320x240, subsampling=4** | 256 ms | 33 ms (350 kb/s) | 204 ms (2000 kb/s) |
| **Gesture recognition** | | | |
| **640x480** | 33 ms | | |
| **320x240, subsampling=2** | 10 ms | | |
| **320x240, subsampling=4** | 9 ms | | |

*Table 35: Overall processing times and framerates for different client/server setups*

| | Processing time for full PDA processing | Processing time for shared processing | Processing time for external processing |
|---|---|---|---|
| **Ms/frame** | | | |
| **640x480** | 1101 ms | 301 ms | 816 ms |
| **320x240, subsampling=2** | 288 ms | 89 ms | 204 ms |
| **320x240, subsampling=4** | 307 ms | 84 ms | 204 ms |
| **Frames/s** | | | |
| **640x480** | 0.91 | 3.32 | 1.23 |
| **320x240, subsampling=2** | 3.48 | 11.27 | 4.9 |
| **320x240, subsampling=4** | 3.26 | 11.93 | 4.9 |

# 7 Summary & Prospects

The design of an efficient system for applications with high demands for real-time performance requires the selection of an appropriate system architecture and incorporated hardware and software components. In order to make such choices, it is imperative to have a detailed knowledge of the application's computational demands. Furthermore, for data-intensive applications the influence of memory accesses has to be taken into account. This work presents a profiling methodology that provides this information. This information includes clock cycles, numerous memory access statistics and several special results providing memory access optimization and data placement in memory. The results are delivered on a detailed level for source code functions and data areas, such as global variables. This work shows how profiling can be integrated into the design flow. The tool aids the designer in making the right decision at each step of the design, including hardware/software partitioning, optimization of components and system scheduling. The profiling methodology has been applied in the development of a software solution and a hardware/software system for real-time video decoding.

Besides performance and memory profiling, the profiler has been extended with energy estimation on a function-accurate level. The development of the underlying power model of a processor is described in detail and can be used as a general approach for model generation. Energy profiling can be used to identify hot-spots in an application. These hot-spots are the most promising candidates for energy consumption reduction. Furthermore, the results can be used to inspect the instruction set of a processor and the influence of each instruction on the energy consumption. This information aids the designer in developing an energy-efficient instruction set for customizable processor architectures.

## 7.1 Comparison with Existing Tools

Existing profiling tools can not deliver the broad profiling results required for the comprehensive optimization and exploration tasks necessary during the design of embedded systems. Gprof is a very useful tool for initial software analysis, but the results are inaccurate due the source code instrumentation and the sampling-based profiling method. Similarly, armprof can also be used only for a rough estimation of the cycle distribution across different parts of the software, and the tool is restricted to the ARM architecture. The same applies for VTune, which is restricted to Intel processors, although the profiling results are very detailed. In addition to the performance results, VTune also provides some information about cache activity, such as cache misses and hits. A far more detailed memory analysis can be performed with the ATOMIUM tool suite. This provides information on memory accesses for each function and variable in the code. The tool is a pure memory profiler and does not deliver any timing information. For the profiling, a generic processor and a flat memory architecture are used, therefore the results are usable only in an early design phase, as the tool cannot reflect the influence of the target architecture. Power consumption estimation is targeted by the JouleTrack tool. Here, measurement-based power models were created for two specific processors. This tool provides results only for the entire program, not for every function.

PowerEscape, which is based on the ATOMIUM tools, offers the most comprehensive profiling. It extends the system architecture model with parameterizable components, such as customizable register files and user-defined memory architecture. In addition to memory analysis, the tool also incorporates timing and power consumption models. However, the system architecture is still a generic architecture, which restricts the accuracy of the profiling. Furthermore, as of 2006 PowerEscape is no longer available.

In contrast to the aforementioned techniques and to overcome their restrictions, the approach presented in this work combines fast, accurate and comprehensive profiling. The trade-off between a decent simulation time and a sufficient level of accuracy is reached by using a tracing-based profiling approach that applies cycle-accurate simulators. In order to target a broad range of processors, a well-defined and simple interface is established for interconnection with the processor simulator. Thus any cycle-accurate processor model can be used, as long as it provides access to basic runtime information such as the program counter, cycle counter and memory busses. The profiler is independent of the application's source code, which leads to higher accuracy as compared to instrumentation-based tools. This additionally allows the profiling of applications that are only available as binaries.

The instruction-level power model presented in this work allows a detailed and measurement-based profiling. However, the current implementation of the energy model is restricted to simple RISC processor architectures with a single pipeline. Complex architecture concepts, such as VLIW or superscalar execution units, are not considered in this work and would require major extensions to the proposed modeling, for example by incorporating FLPA methodologies. However, these architectures are currently not wide-spread on the embedded RISC processor market. This is due to the large overhead they involve; for example, features such as out-of-order execution increase the required die area significantly.

## 7.2 Prospects

The profiling tool described here is still missing some features. Callgraph-based profiling, an essential feature of gprof, is not included in the profiler. Instead, a straightforward approach for accumulating the results of called functions is applied, which allows only a restricted view of the clock cycle distribution from a hierarchical view. This restriction is acceptable only for optimization purposes, as hot spots are still visible.

The optimization steps presented in this work need to be applied manually by the software developer. For complex applications especially, this can be a cumbersome task. Therefore, the profiler should be integrated into the compiler suites. The compiler can then be instructed to perform the code modifications automatically based on the profiling results.

Areas for future development also include the extension of the power model to other processor architectures and applying the methodology to other application fields in order to show its versatility. Furthermore, power models for the memory architecture could be included in order to reflect their influence on overall energy consumption. The retargeting of the power model can be simplified by separating the CPU core power model from the cache modeling. Caches could be modeled by means of the CACTI [96] cache model, which offers very detailed results and can be used for a wide range of technology feature sizes. On-chip SRAM can be incorporated into the model by adapting the CACTI tool, and external DRAM devices can be modeled with datasheet based models [59].

# 8 Appendix

## 8.1 Detailed and Comprehensive Profiling Results

### 8.1.1 H.264/AVC Encoder/Decoder

Figure 53 shows the callgraph of the software implementation of an H.264/AVC Baseline decoder. This implementation has been used within the case study described in Section 6.1.

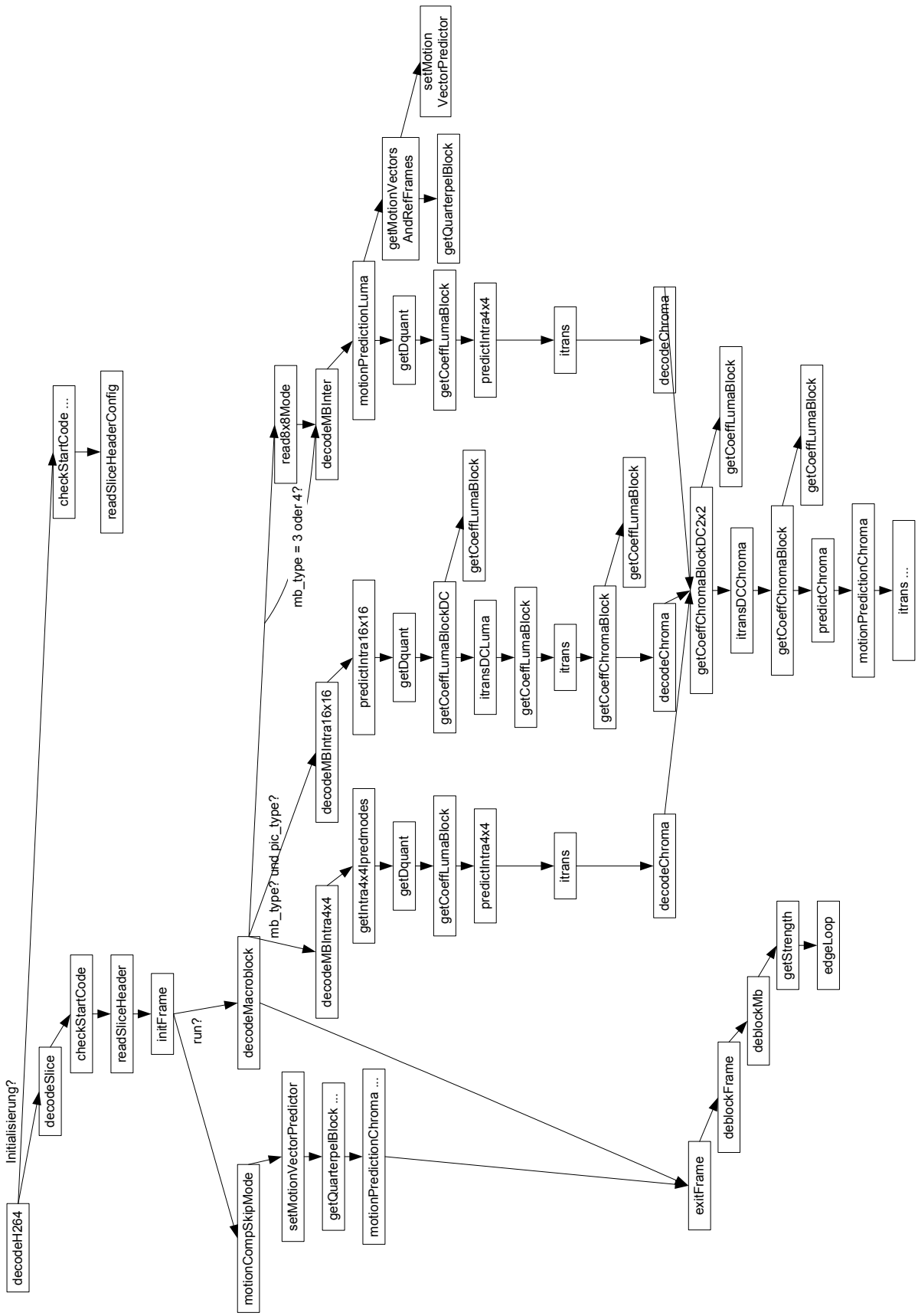



*Figure 53: Callgraph of an H.264/AVC baseline decoder*

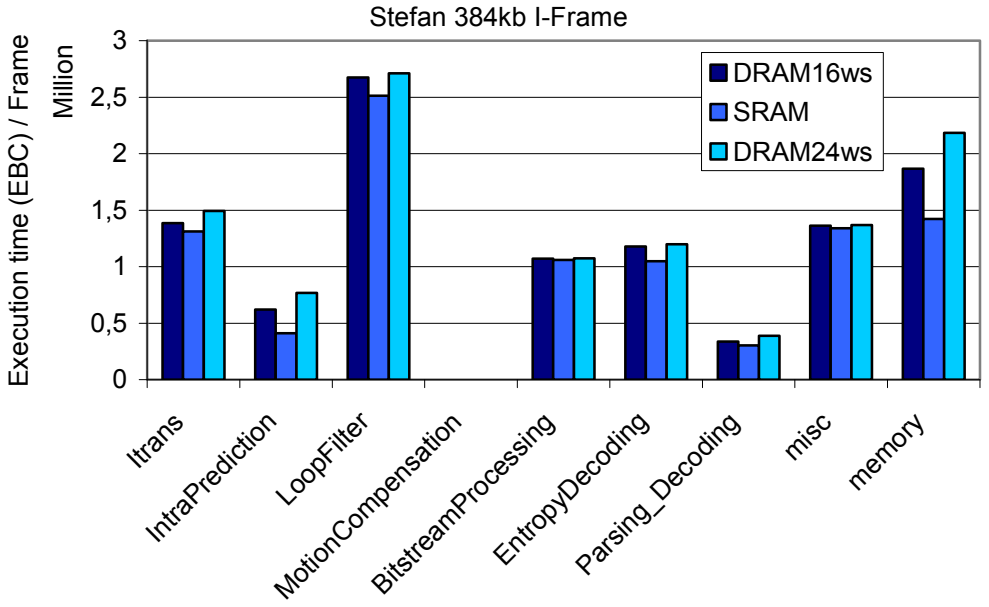



*Figure 54: Function group analysis for worst case I-frame with different memory types for sequence "Stefan 384kb"*

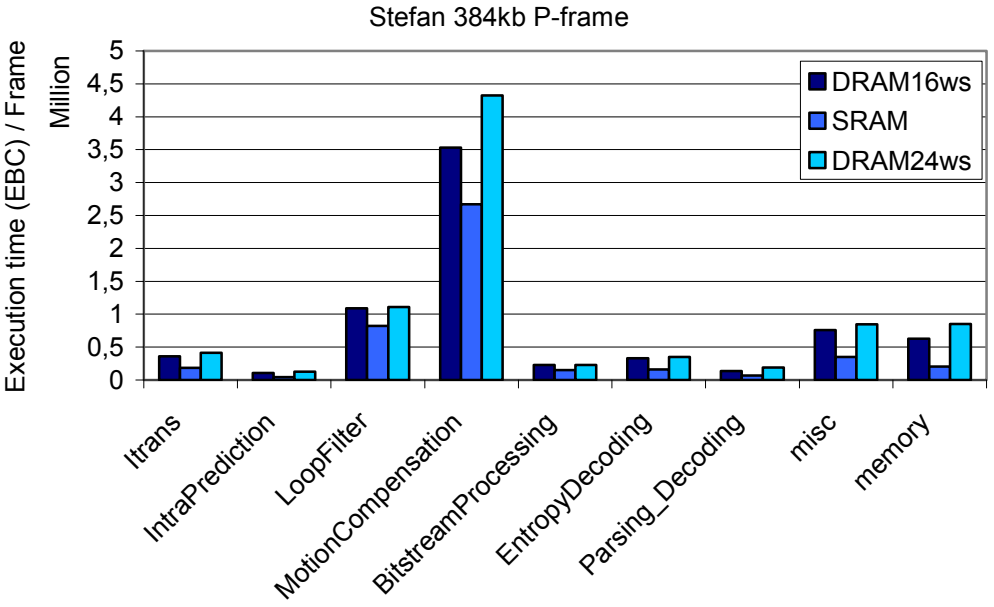



*Figure 55: Function group analysis for worst case P-frame with different memory types for sequence "Stefan 384kb"*

### 8.1.2 Function Group Analysis for I- and P-Frames

Function group analysis is based on the grouping of all incorporated functions into the following groups:

- Itrans — inverse H.264/AVC transformation
- Intraprediction — H.264/AVC intra prediction algorithms
- Loopfilter — frame deblocking filter including all filters
- Motioncompensation — motion compensation for P-frames
- Bistreamprocessing — getbits, showbits, flushbits
- Entropy Decoding — CAVLC decoding
- Parsing / Decoding — high level parsing of bitstreams
- Memory — memory transfer functions like memcpy, memset
- Misc — miscellaneous functions

The results were generated for I- and P-frames.

In the following diagrams a side by side comparison of I- and P-picture coding types with their needed external bus cycles for the different function groups and the used memory architecture is depicted. The data has been generated using the worst case pictures of the corresponding bitstream. Execution time is measured in number of external bus cycles as provided by MEMTRACE in conjunction with the ARMulator.

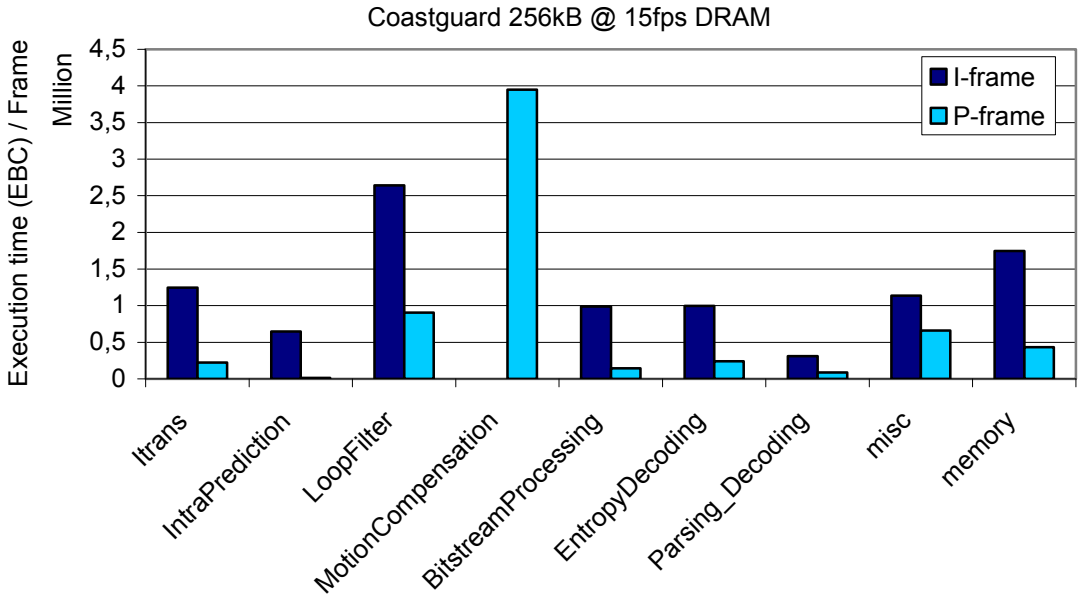



*Figure 56: Function group analysis for worst case I- and P-frame with DRAM16 memory types for sequence "Coastguard 256kb"*

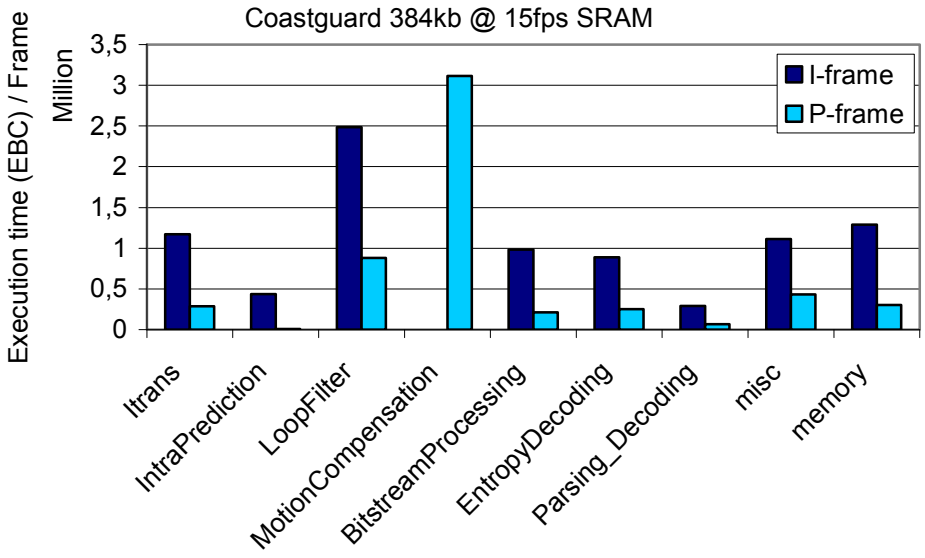



*Figure 57: Function group analysis for worst case I- and P-frame with SRAM types for sequence "Coastguard 384kb"*

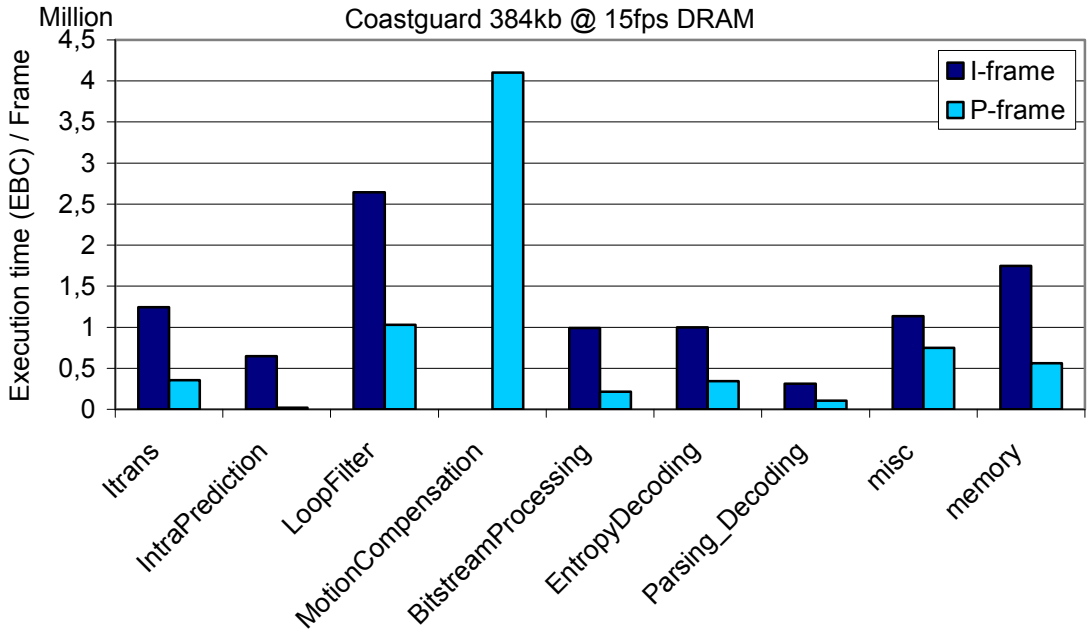



*Figure 58: Function group analysis for worst case I- and P-frame with DRAM16 memory types for sequence "Coastguard 384kb"*

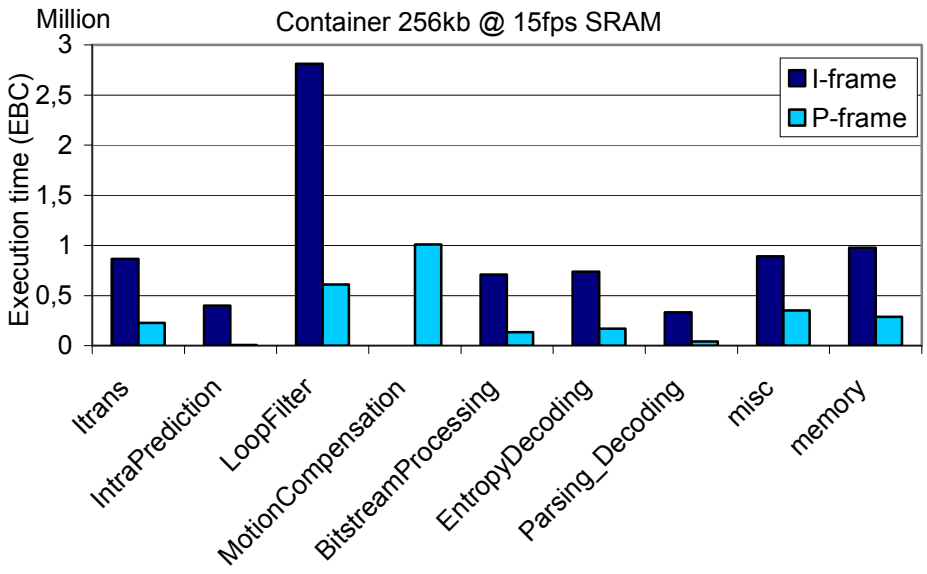



*Figure 59: Function group analysis for worst case I- and P-frame with SRAM types for sequence "Container 256kb"*

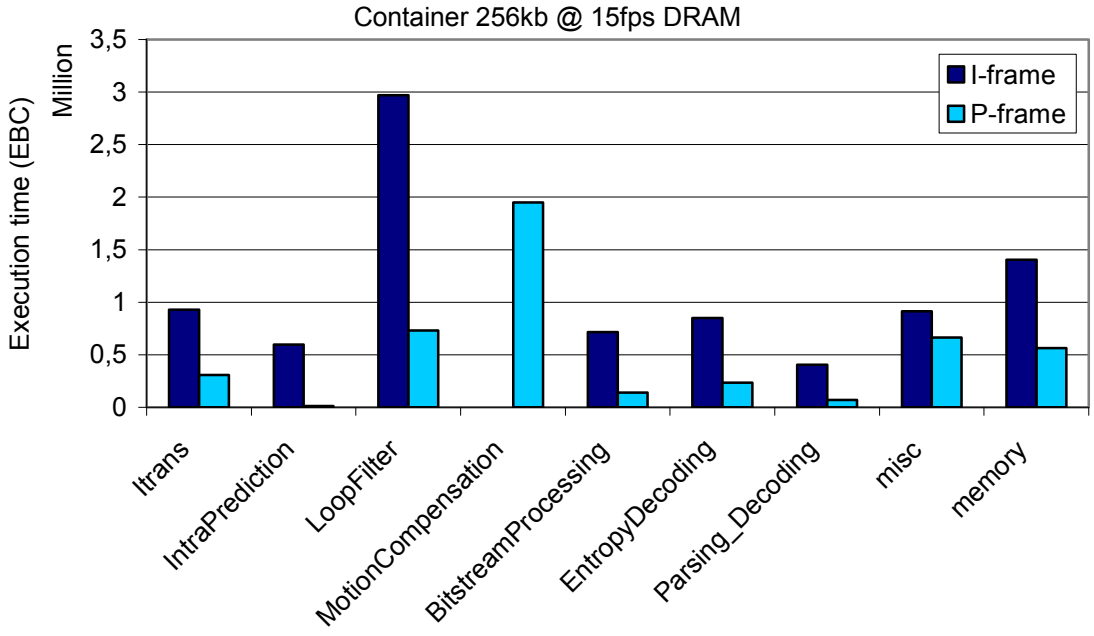



*Figure 60: Function group analysis for worst case I- and P-frame with DRAM16 memory types for sequence "Container 256kb"*

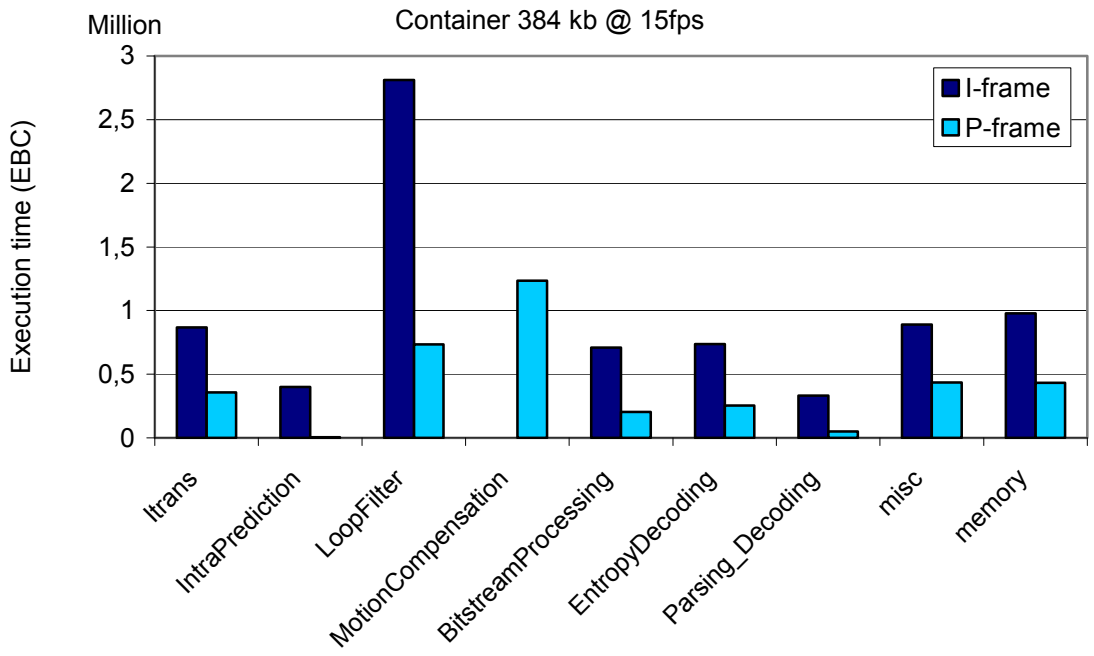



*Figure 61: Function group analysis for worst case I- and P-frame with SRAM types for sequence "Container 384kb"*

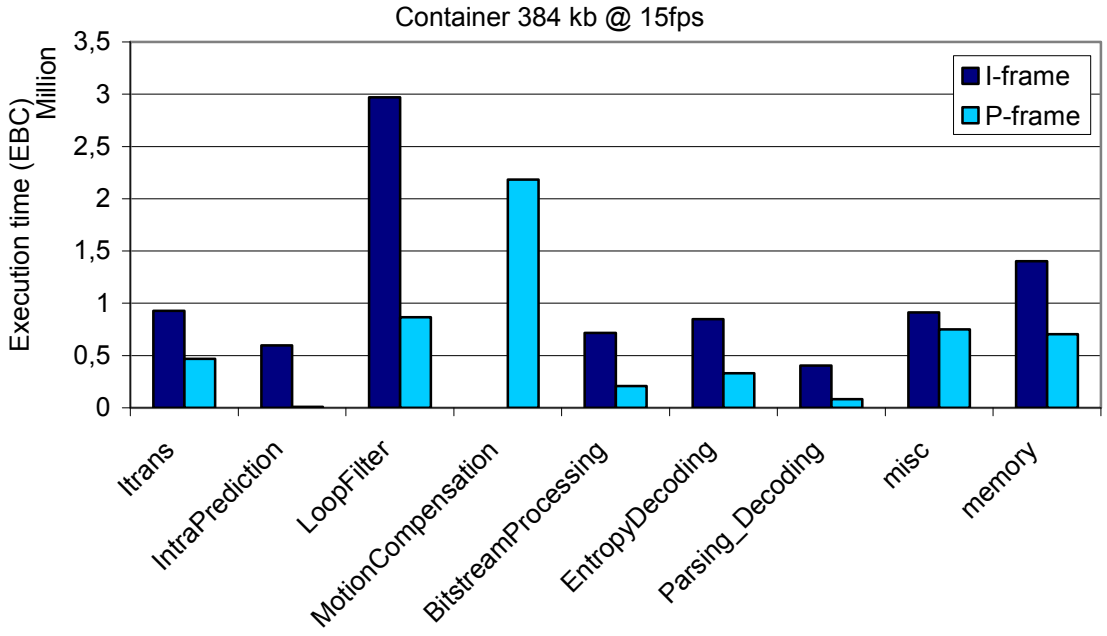



*Figure 62: Function group analysis for worst case I- and P-frame with DRAM16 memory types for sequence "Container 384kb"*

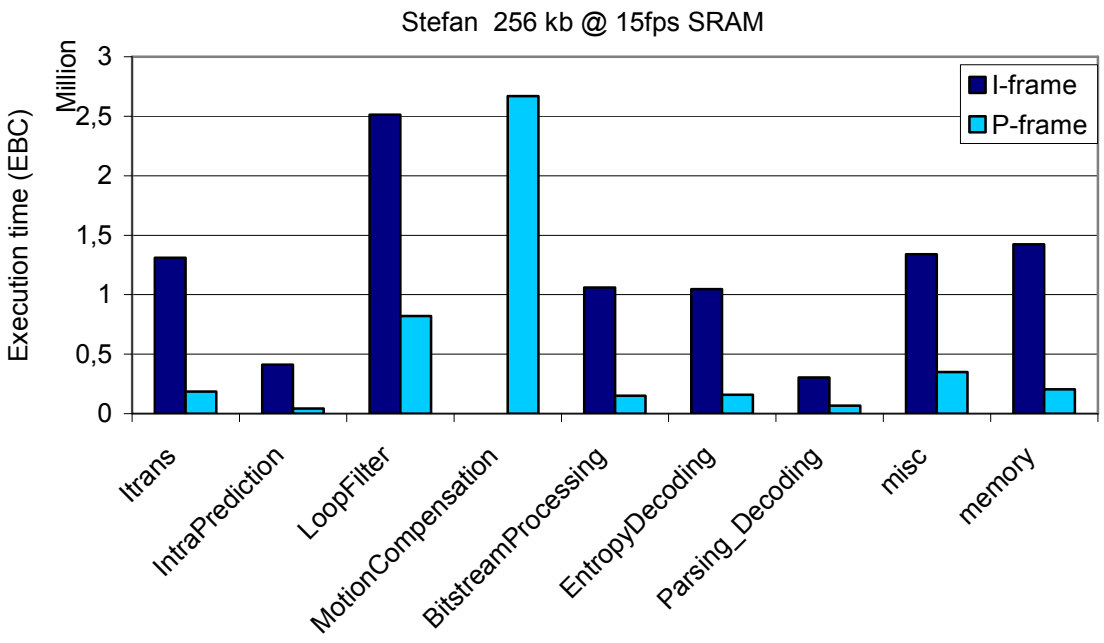



*Figure 63: Function group analysis for worst case I- and P-frame with SRAM types for sequence "Stefan 256kb"*

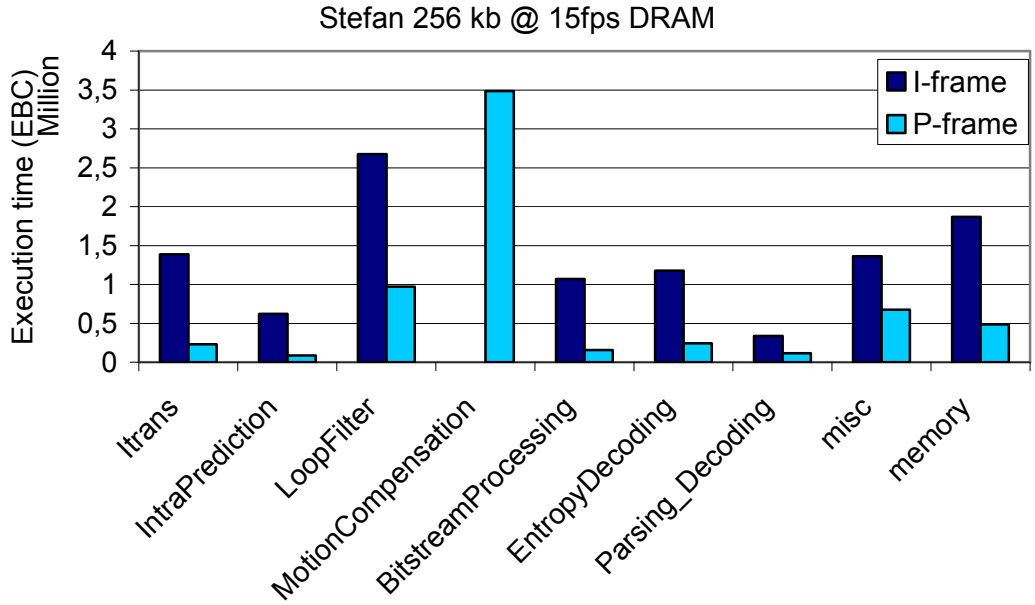



*Figure 64: Function group analysis for worst case I- and P-frame with DRAM16 memory types for sequence "Stefan 256kb"*

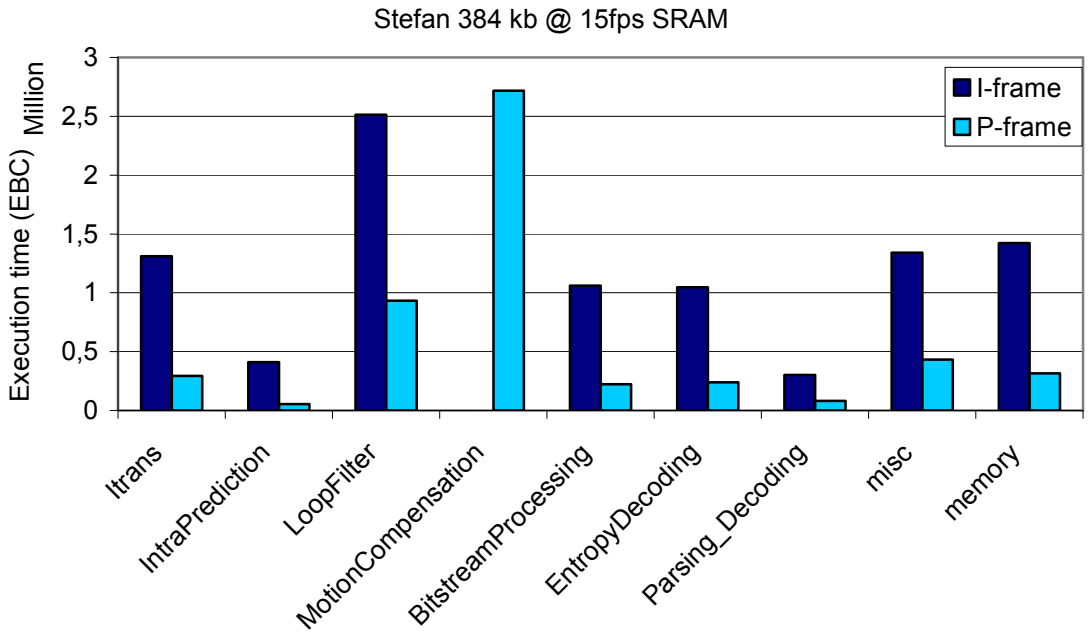



*Figure 65: Function group analysis for worst case I- and P-frame with SRAM types for sequence "Stefan 384kb"*

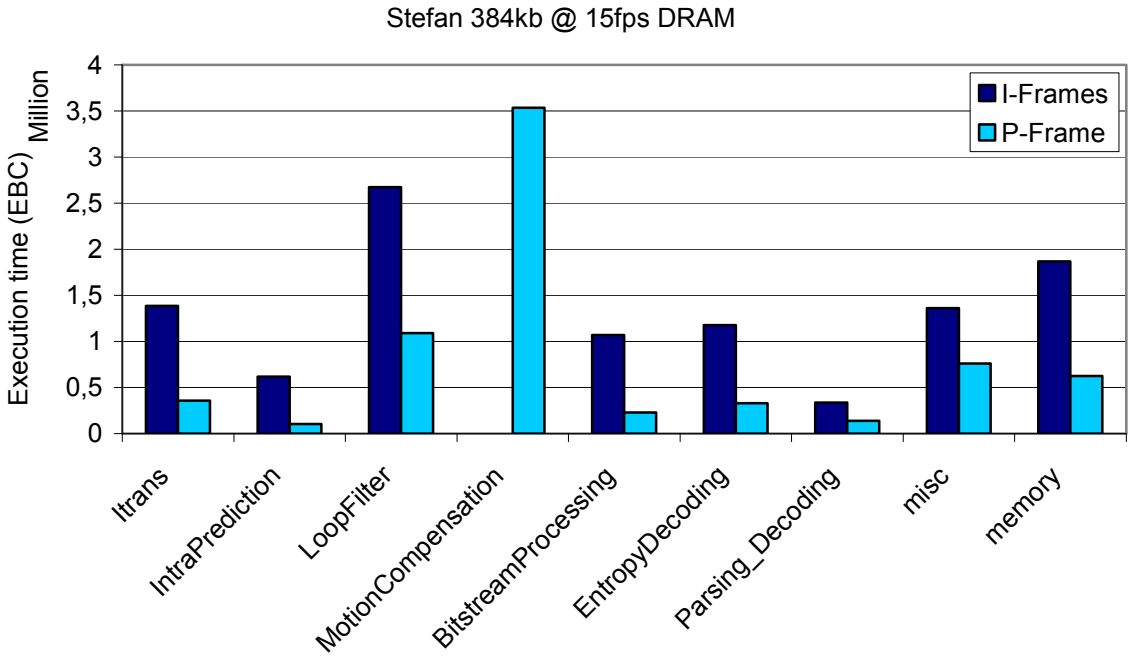



*Figure 66: Function group analysis for worst case I- and P-frame with DRAM16 memory types for sequence "Stefan 384kb"*

### 8.1.3 Cycles per Frame Analysis

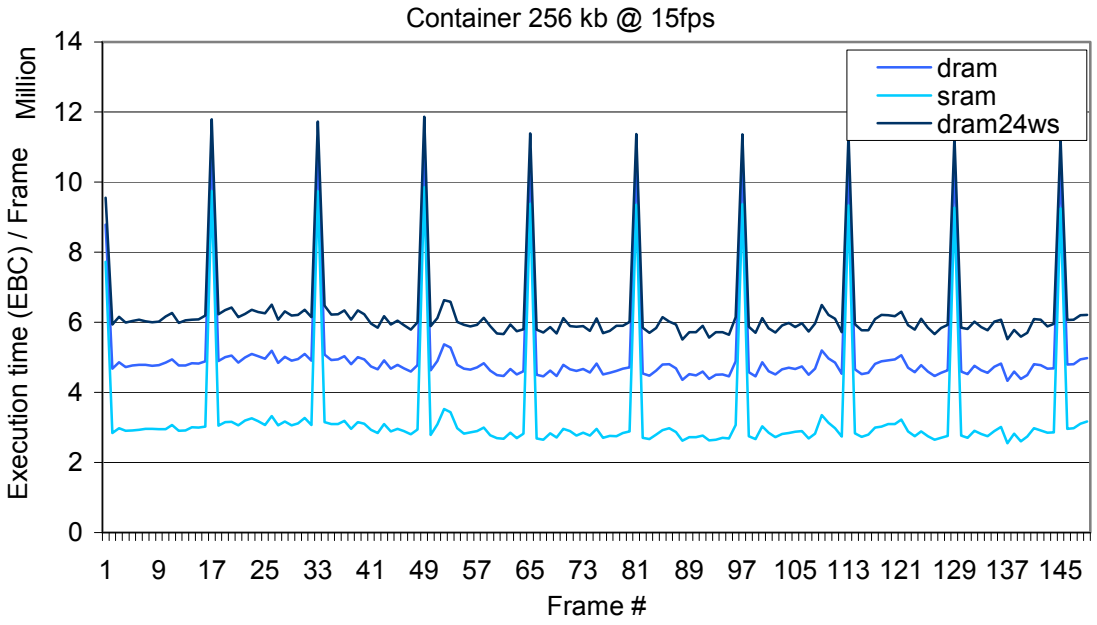



*Figure 67: Cycles per frame analysis for different memory types for sequence "Container 256 kb"*

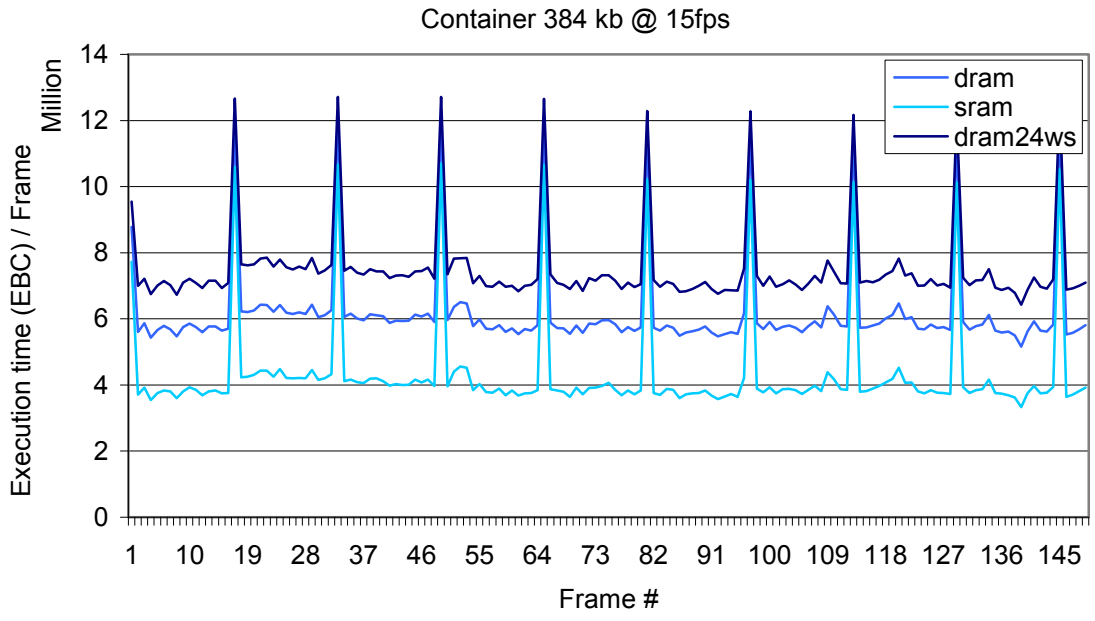



*Figure 68: Cycles per frame analysis for different memory types for sequence "Container 384 kb"*

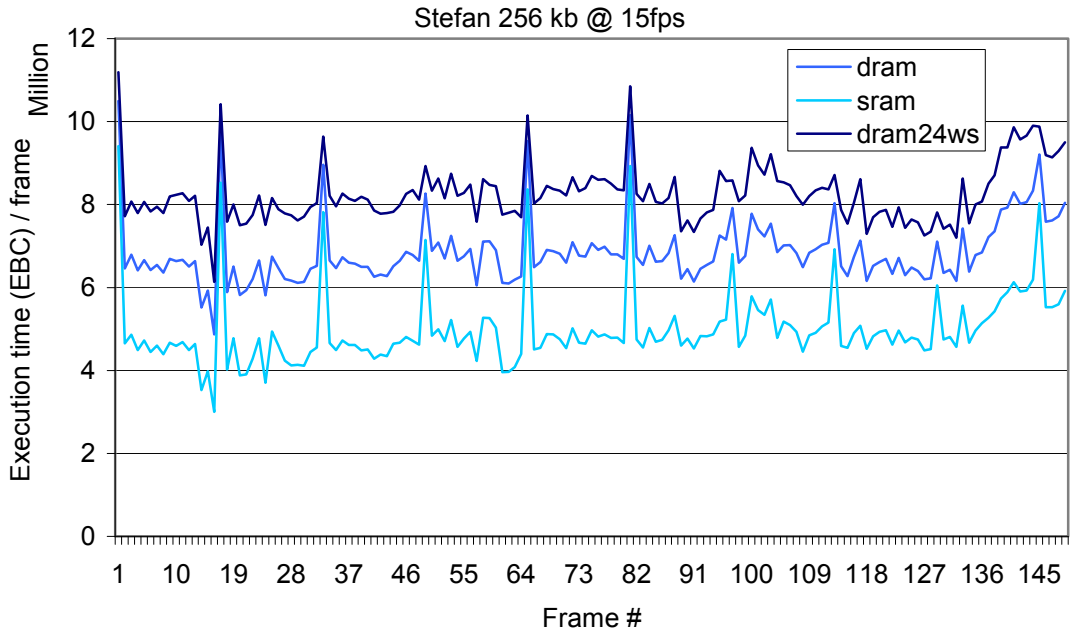



*Figure 69: Cycles per frame analysis for different memory types for sequence "Stefan 256 kb"*

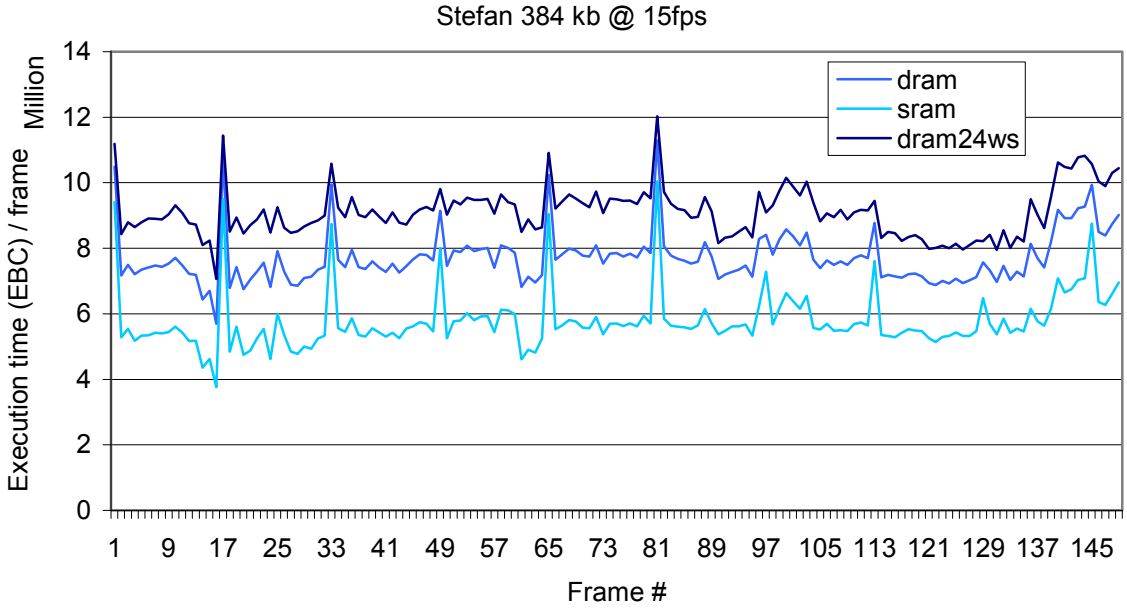



*Figure 70: Cycles per frame analysis for different memory types for sequence "Stefan 384 kb"*

### 8.1.4 Usage of ARM11 SIMD Instructions

#### 8.1.4.1 I-Trans

The function group I-trans combines all modules needed for the inverse transformation including inverse quantization. The following table gives an overview over the implemented functions and their potential for optimization using ARM11 SIMD instructions. For the following functions the reorganization overhead for data formatting is too high to make an SIMD implementation feasible:

GetCoeffChromaBlockDC2x2, GetCoeffLumaBlock, GetCoeffLumaBlockDC, ItransDCChroma

*Table 36: Functions of the I-trans group*

| Function name | Benefit / used instructions | Optimization factor | Implementation difficulty |
|---|---|---|---|
| **GetCoeffChromaBlock** | Run / level decoding | 1 | |
| **Itrans** | First function part: 2<br>Second function part:1 | 1.5 | Low - medium |
| **ItransDCLuma** | First function part :2<br>Second function part : 1 | 1.5 | Low - medium |

#### 8.1.4.2 Intraprediction

The function group Intraprediction combines all modules needed for the prediction of pixels of intra coded blocks . The following table gives an overview over the implemented functions and their potential for optimization using ARM11 SIMD instructions. For the following functions the reorganization overhead for data formatting is too high:

PredictIntra4x4_DIAGDOWNLEFTUR, PredictIntra4x4_DIAGDOWNRIGHT, PredictIntra4x4_HOR, PredictIntra4x4_HORDOWN, PredictIntra4x4_HORUP, PredictIntra4x4_VER, PredictIntra4x4_VERTLEFT, PredictIntra4x4_VERTRIGHT PredictIntra4x4_VERTRIGHTUR, PredictIntra4x4_DC128, PredictIntra4x4_DCL

*Table 37: Functions of the intraprediction group*

| Function name | Benefit / used instructions | Optimization Factor | Implementation difficulty |
|---|---|---|---|
| **PredictIntra4x4_DC** | Using UADD8to16 | 1.1 | Low |
| **PredictIntra4x4_DCU** | Using UADD8to16 | 1.1 | Low |
| **PredictIntra4x4_DIAGDOWNLEFT** | reg. width not sufficient | 1 | |
| **PredictChroma** | LD/ST overhead too big | 1 | |
| **GetIntra4x4Ipredmodes** | LD/ST overhead too big | 1 | |
| **PredictIntra16x16** | LD/ST overhead too big | 1 | |

### 8.1.4.3 Loopfilter

The function group Loopfilter combines all modules needed for the deblocking of a picture after the decoding process. The following table gives an overview over the implemented functions and their potential for optimization using ARM11 SIMD instructions.

*Table 38: Functions of the loop-filter group*

| Function name | Benefit / used instructions | Optimization factor | Implementation difficulty |
|---|---|---|---|
| **DeblockFrame** | Wrapper function | 1 | |
| **DeblockMB** | Wrapper function | 1 | |
| **DeblockMB_Intra** | Wrapper function | 1 | |
| **EdgeLoopC_N** | Using UADD8to16 | 2 | Complex |
| **EdgeLoopC_S** | Using UADD8to16 | 2 | Complex |
| **EdgeLoopY_N** | Using UADD8to16 | 2 | Complex |
| **EdgeLoopY_S** | Using UADD8to16 | 2 | Complex |
| **GetStrength_QP_MB** | Vectorization, algorithm optimization without SIMD instructions | 1.2 | Complex |

### 8.1.4.4 Motion Compensation

The function group MotionCompensation combines all modules needed for the reconstruction and filtering of a predictive coded picture. The following table gives an overview over the implemented functions and their potential for optimization using ARM11 SIMD instructions. For the following functions the reorganization overhead for data formatting is too high:

SetMotionVectorPredictorP, GetMotionVectorsAndRefFrames, GetMotionVectorsAndRefFrames16x16, GetMotionVectorsAndRefFrames16x8, GetQuarterpelBlock8_xy

*Table 39: Functions of the motion compensation group*

| Function name | Benefit / used instructions | Optimization factor | Implementation difficulty |
|---|---|---|---|
| **Copypadblock8** | LD/ST overhead too high | 1 | |
| **Copypadblock** | LD/ST overhead too high | 1 | |
| **GetQuarterpelBlock_xy** | Algorithm optimization on C source code level | 1.2 | Complex |
| **ProcessInterP16x16** | Function caller | 1 | |
| **ProcessInterP16x8** | Function caller | 1 | |
| **ProcessInterP4x4** | Function caller | 1 | |
| **ProcessInterP8x16** | Function caller | 1 | |
| **MotionCompSkipMode** | LD/ST oriented, parallelization | 1 | |
| **MotionPredictionChroma** | Parallelization of 2 x 2x2 predictions | 1.2 | Complex |

#### 8.1.4.5 BitstreamProcessing

The function group BitstreamProcessing combines all modules needed for the bitwise stream processing an H.264/AVC access unit. Due to the nature of the functions it makes nearly no sense to use any SIMD instruction for optimization purposes. The following table is provided to give an overview of the used functions in the according function group. The following list provides an overview of the used functions in this function group:

BitsLeft, ByteAligned, FlushBits, GetBits, GetOneBit, ShowBits, ShowBitsN, GetStrength_QP_MB

#### 8.1.4.6 EntropyDecoding

The function group EntropyDecoding combines all modules needed for the VLC stream processing an H.264/AVC access unit. Due to the nature of the functions it makes nearly no sense to use any SIMD instruction for optimization purposes. The following list provides an overview of the used functions in this function group:

GetVLCSymbol, GetVLCSymbol_Slow, CodeFromBitstream2d, PredictNnz, ReadCoeffBlockCAVLC, ReadLevelVLC0, ReadLevelVLCN, ReadLongRuns, ReadNumCoeffTrailingOnes, ReadNumCoeffTrailingOnesChromaDC, ReadShortRuns, ReadTotalZeros, ReadTotalZerosChromaDC

#### 8.1.4.7 Parsing_Decoding

The function group Parsing_Decoding combines all modules needed for the high level stream processing of an H.264/AVC access unit. Due to the nature of the functions it makes nearly no sense to use any SIMD instruction for optimization purposes. The following list provides an overview of the used functions in this function group:

ParsePictureParameterSet, DecodeChroma, DecodeMBInter, DecodeMBIntra16x16, DecodeMBIntra4x4, DecodeMacroblock, GetDquant, Predblock4, Read8x8Mode

## 8.2 MEMTRACE Implementation Details

### 8.2.1 Block Diagrams of MEMTRACE Internals

The following block diagrams show excerpts of the function callgraphs of the MEMTRACE source code. Figure 71 depicts the callgraph of the MEMTRACE CLI applications. It combines the CLI frontend with parts of the backend, e.g. for program information acquisition.

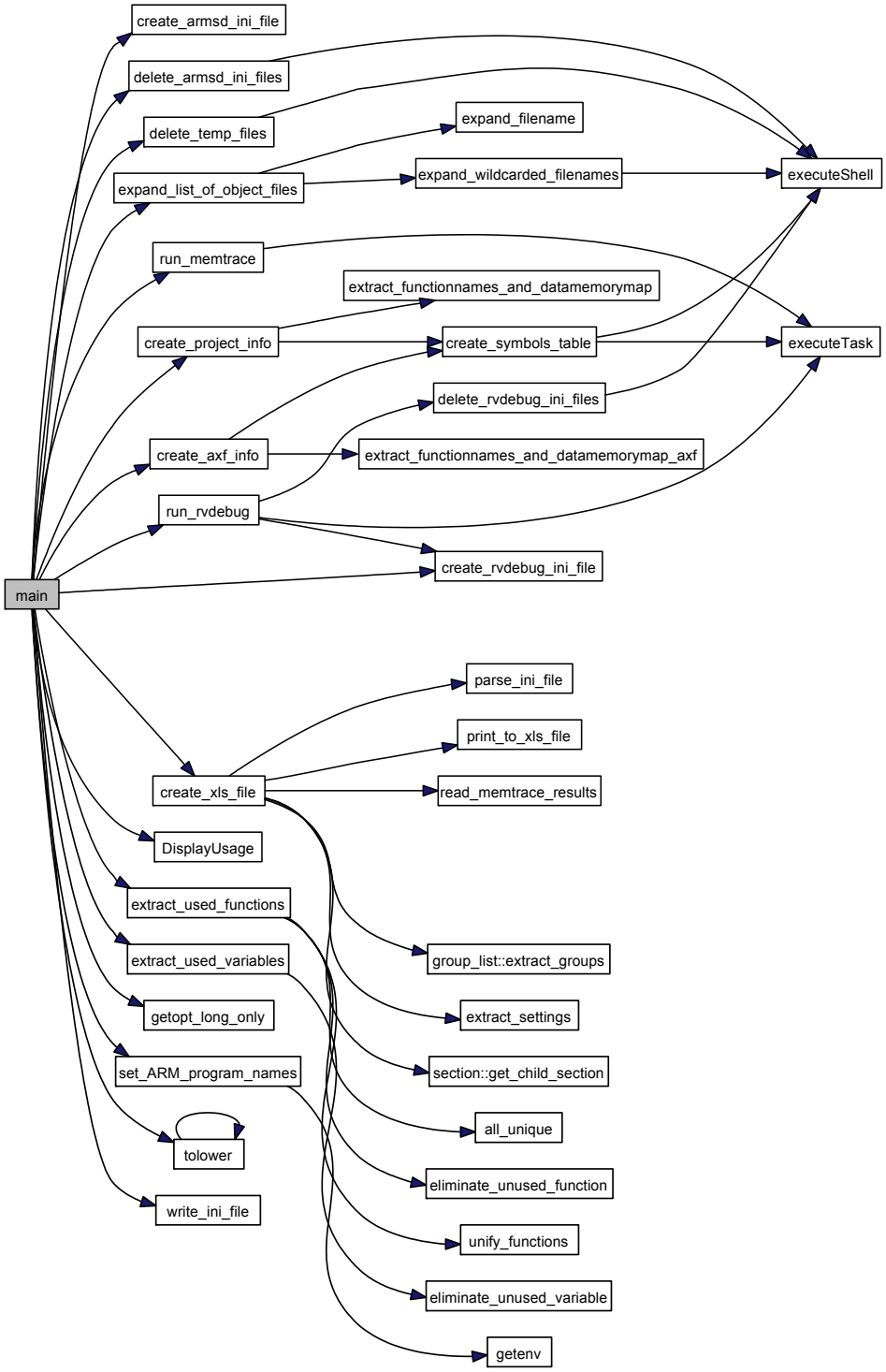



*Figure 71: Callgraph of the main function of the MEMTRACE*

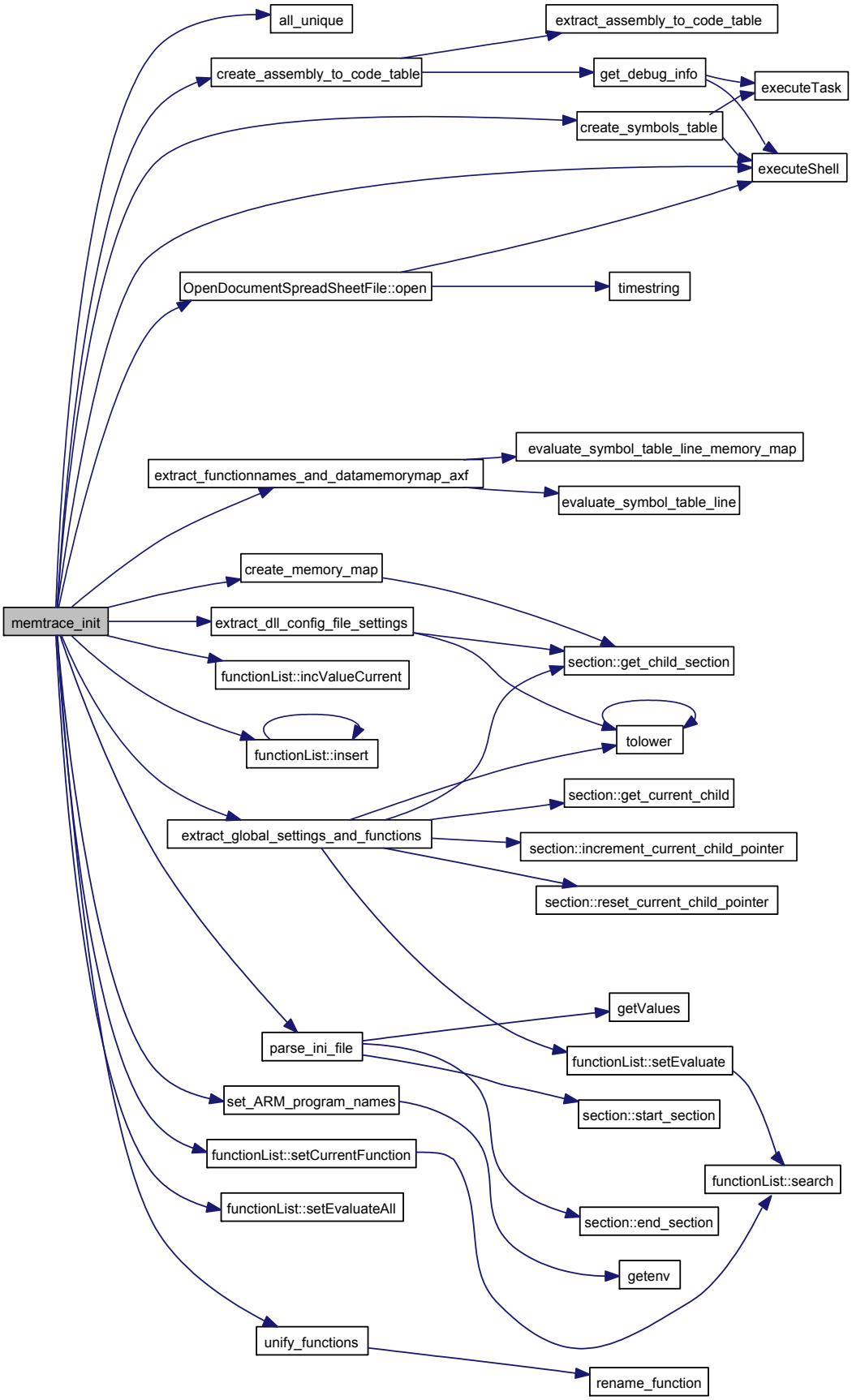



*Figure 72: Callgraph of backend function memtrace_init*

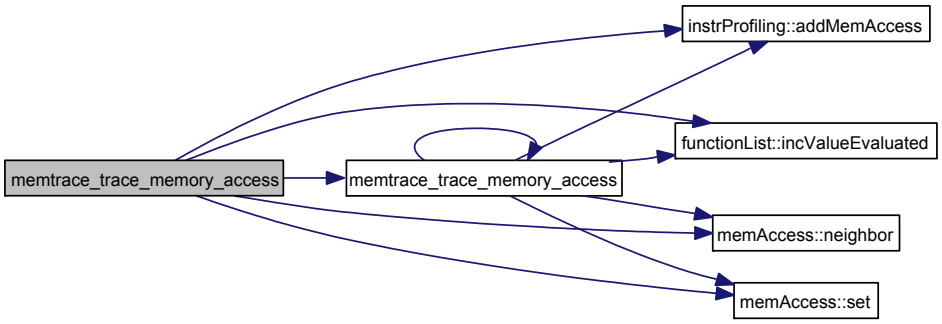



*Figure 73: Callgraph of backend function memtrace_trace_memory_access*

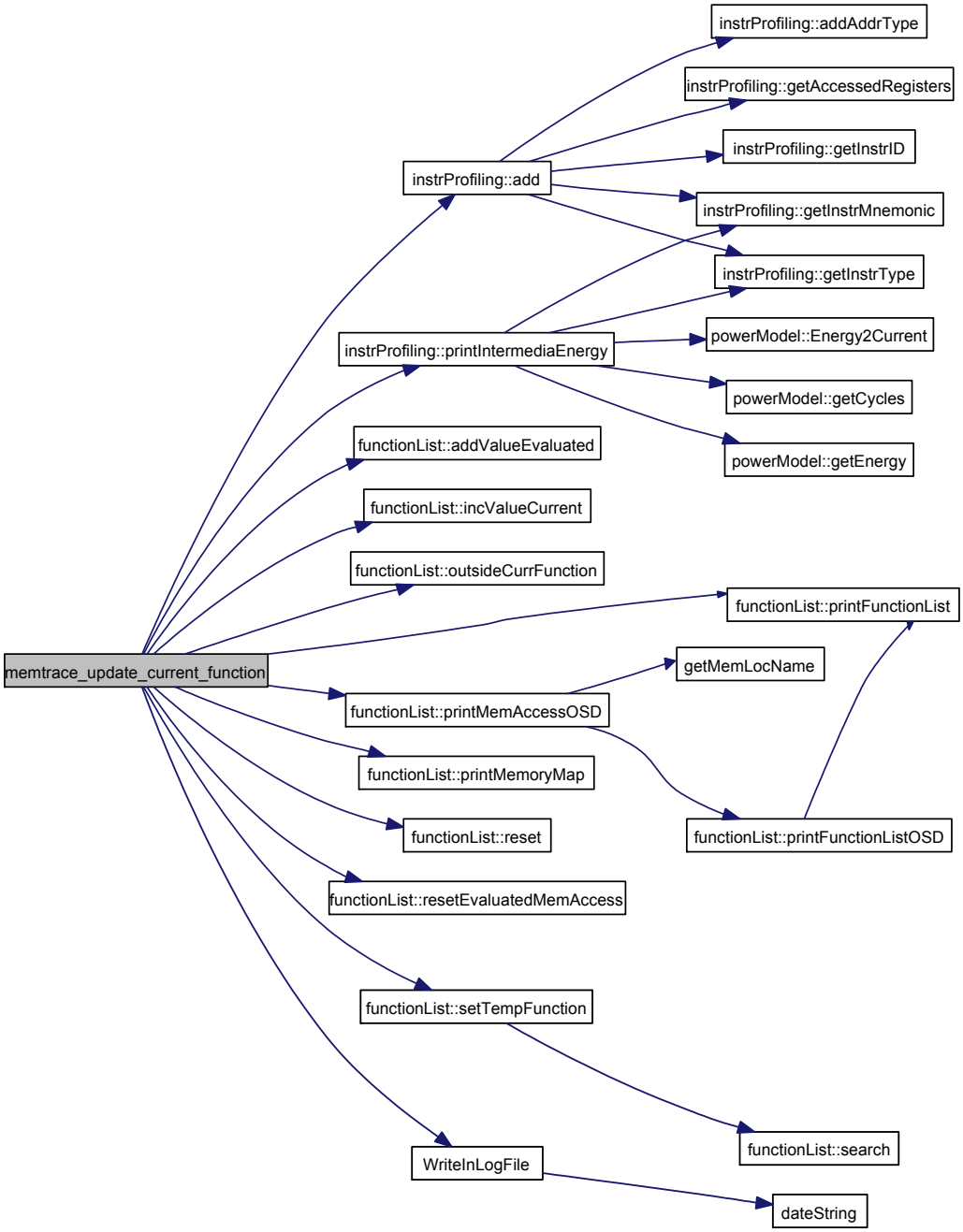



*Figure 74: Callgraph of backend function memtrace_update_current_function*

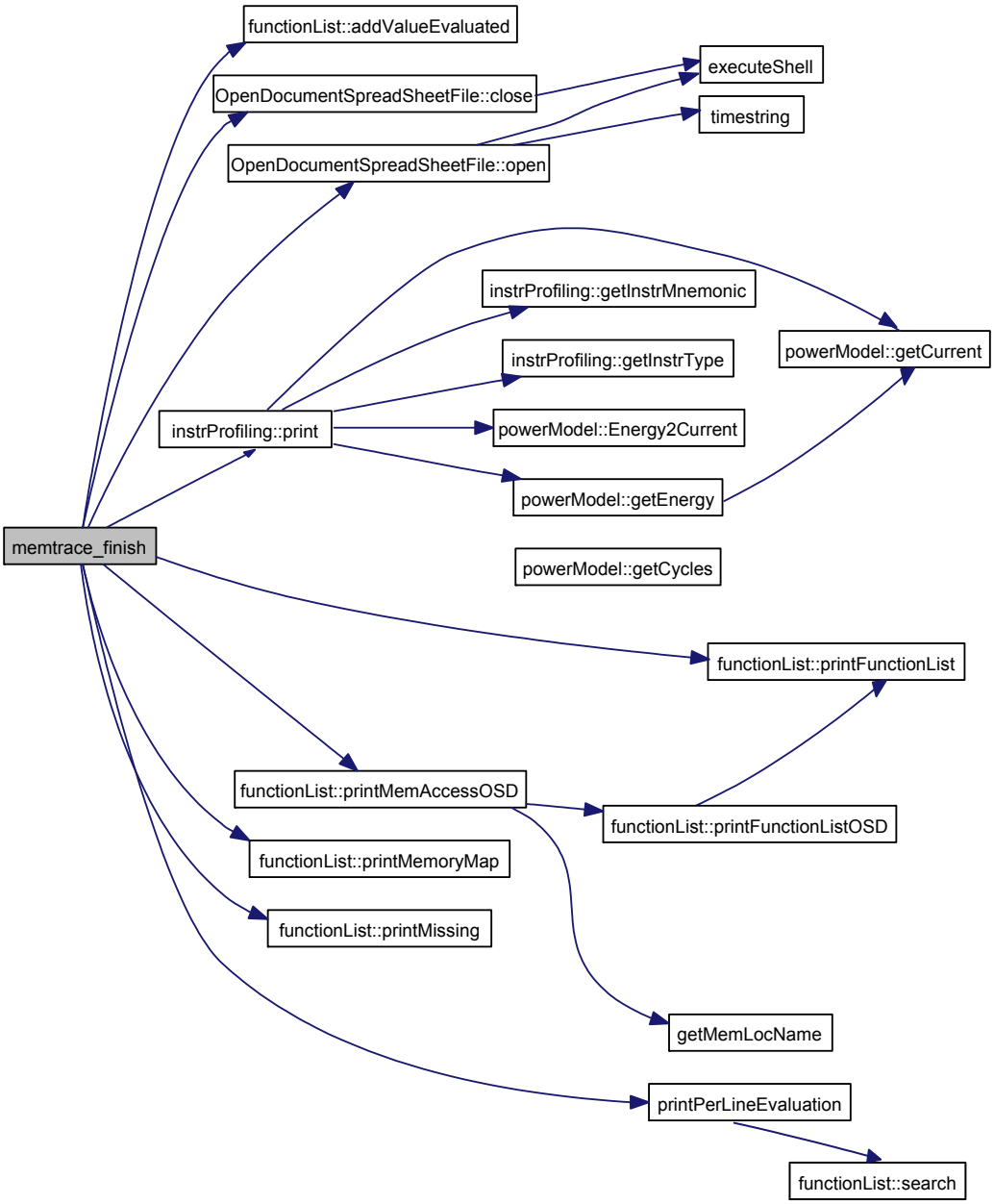



*Figure 75: Callgraph of backend function memtrace_finish*

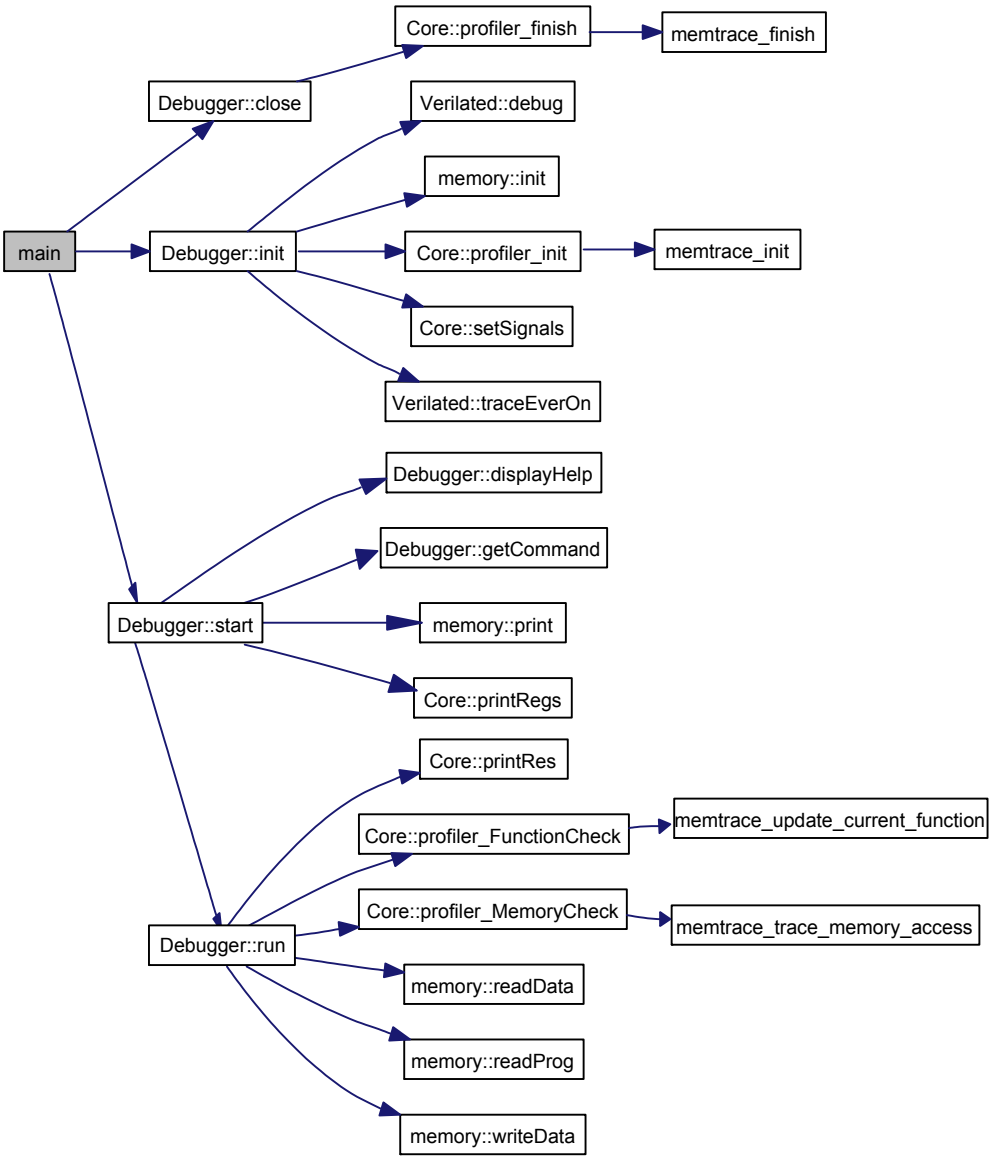



*Figure 76: Callgraph of the MEMTRACE debugger*

### 8.2.2 Screenshots of the Graphical User Interface

Figure 77 shows a screenshot of the main window of the GUI. The upper part of the window provides the files settings for the MEMTRACE project respectively configuration file, the executable (axf) file and analysis and the postprocessing output files. The tab area shows the initializations tab, where on the left side the object files or directories can be specified and the symbol extraction can be started (“extract program info”). The right side shows the program information, which includes the functions, variables and other memory areas found the object files and the setting area for the memory layout. Furthermore the split step can be enabled and a split function be chosen.

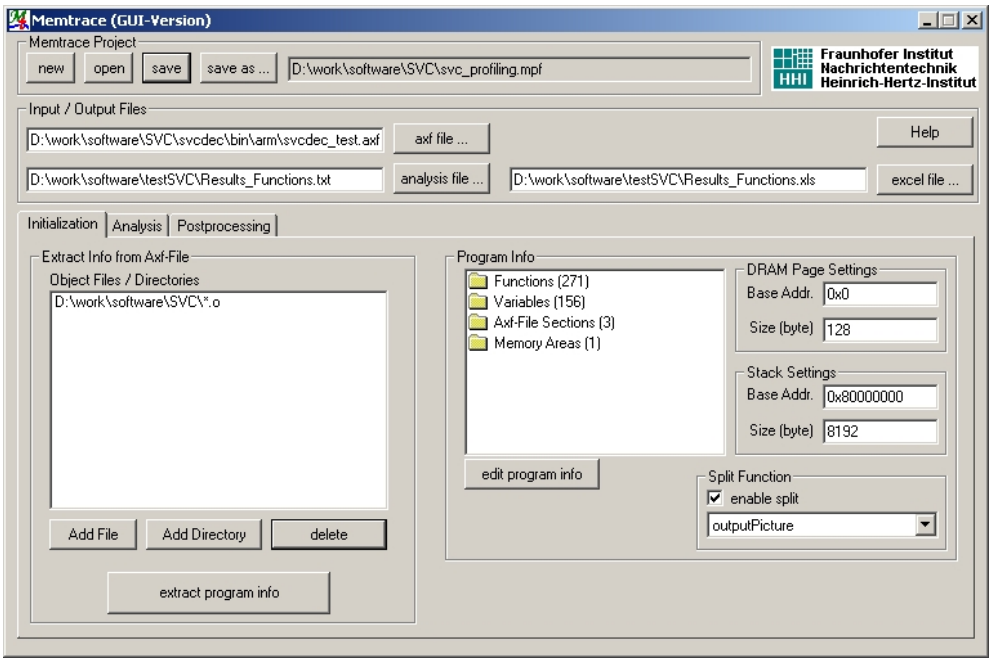



*Figure 77: Graphical user interface to the MEMTRACE tool (initialization tab)*

The lists of function and memory areas can be modified by pressing “edit program info” button activating the window shown in Figure 78. It allows combining functions to functional groups in order to sum their results. Furthermore the results for each function can be manipulated by incorporating a multiplication factor as described in Section 4.3.4.

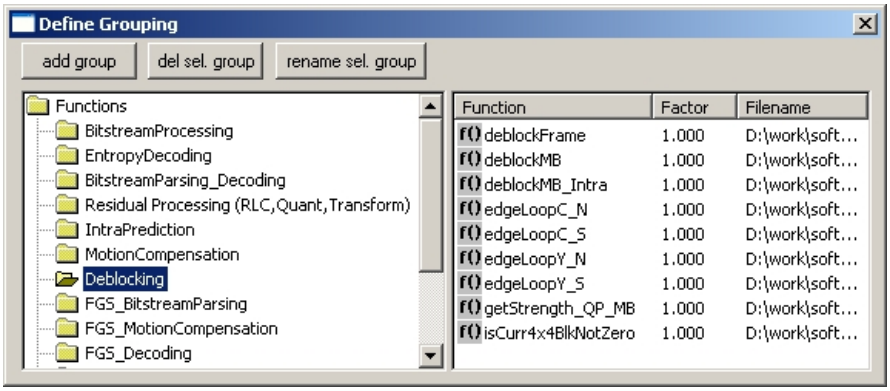



*Figure 78: GUI dialog for viewing and setting function details*

Figure 79 shows the analysis tab where all settings for the simulation run are applied. The system specification includes the processor type and its clock frequency and the divider between bus and processor core speed. Additionally, command-line parameters for the executable (axf) file can be provided.

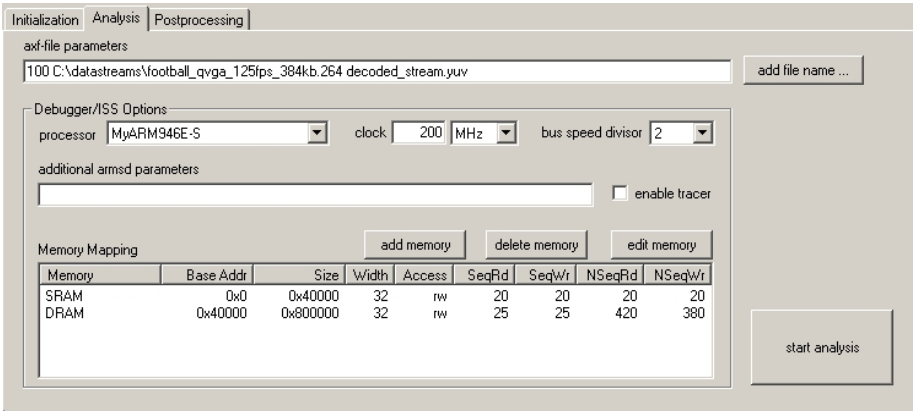



*Figure 79: Graphical user interface to the MEMTRACE tool (analysis tab)*

A memory map is defined by specifying the timing and access information of the different memory regions. A settings dialog shown in Figure 80 is used for this purpose. The memory model allows differentiating between sequential and non-sequential read and write accesses and to specify the width and access type of the according device.

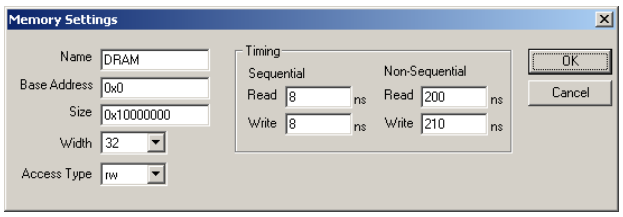



*Figure 80: Memory settings dialog window*

The postprocessing step is controlled with the third tab shown in Figure 81. It allows choosing between creating spreadsheet tables for function and variable analysis results. A table is defined by the table type, the row/column type and the range definitions. These are described in more detail in Section 5.4. Additionally to the command-line interface, the GUI also allows to specify a mathematical function to be applied to the results. For example, a “bandwidth” function transforms the memory access results into bus bandwidth values, depending on the bus speed.

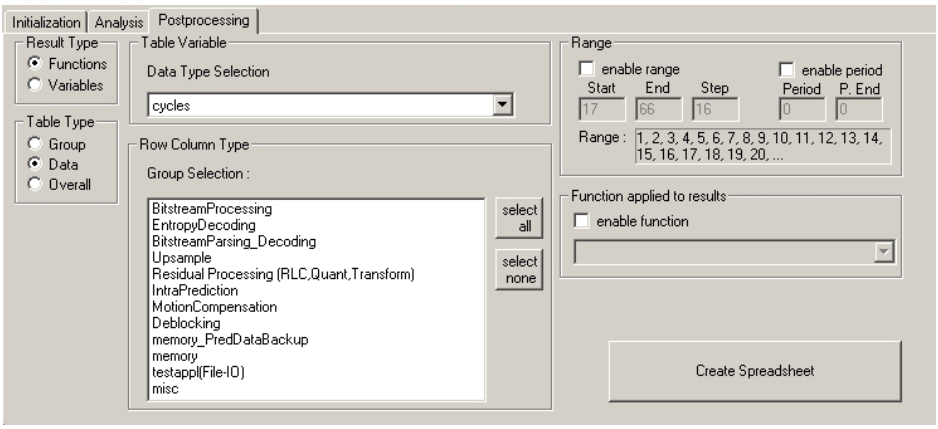



Figure 81: Graphical user interface to the MEMTRACE tool (postprocessing tab)

### 8.2.3 Detailed Power Measurement Results

Table 40 shows the current values, which were measured for the core voltage of the Excalibur device during the execution of the test sequences.

*Table 40: Current values measured for the different instruction sequences*

| Prog | current (mA) | Difference to idle value (mA) |
|---|---|---|
| **Idle** | 262 | 0 |
| **Nop** | 317 | 55 |
| **Mov4** | 323 | 61 |
| **Mov3** | 323 | 61 |
| **Mov2** | 325 | 63 |
| **Mov** | 328 | 66 |
| **Add** | 329 | 67 |
| **Add1a** | 336 | 74 |
| **Add1b** | 317 | 55 |
| **Add1c** | 329 | 67 |
| **Add2** | 334 | 72 |
| **Ld3** | 373 | 111 |
| **Ld3a** | 366 | 104 |
| **Ld4** | 359 | 97 |
| **Ld4a** | 359 | 97 |
| **Ld4b** | 360 | 98 |
| **Ld4c** | 356 | 94 |
| **Mov5** | 317 | 55 |
| **Mov_mod1** | 327 | 65 |
| **Ld2_mov (1.5*exectime)** | 348 | 86 |
| **Ld3_mov** | 363 | 101 |
| **Ld4_mov** | 355 | 93 |
| **Ld5_mov** | 339 | 77 |
| **Ld6_mov** | 343 | 81 |
| **Mov4a** | 323 | 61 |
| **Mov4b** | 317 | 55 |
| **Mov4c** | 329 | 67 |
| **Tst (like mov.c)** | 334 | 72 |
| **Tst4b (1reg like mov4b.c)** | 317 | 55 |
| **Tst4c (1reg like mov4c.c)** | 337 | 75 |
| **Tst4c_2reg** | 337 | 75 |
| **Tst4c_2reg_F_0** | 328 | 66 |
| **Tst4b_2reg** | 317 | 55 |

### 8.2.4 The Configuration File

The profiler is controlled by a configuration file. This file is automatically created during the initialization step. It can be edited by, for example by removing or grouping functions, settings the split flag or adding new memory areas. An example configuration file is given in the following. The comment lines in the beginning of the file describe the syntax.

```
;; This is a generated ini-file for memtrace.
;; Edit this file according to your needs.
;; The file format is similar to the ARM configuration file format
;; as used e.g. in the *.ami and *.dsc files. It is described in
;; the "Debug Target Guide" in Section 4.15.2
;; Currently the following parts are supported:
;;
;; Tag = Value
;; Tag = Value1 Value2
;; Othertag
;; ;; comment line
;; ; commented-out line
;; { MySection = SectionName
;; SectionTag1
;; SectionTag2
;; }
;;
;; IMPORTANT: * All Tag values and Names should only contain al-
pha-numerical symbols and "_"
;;               (and currently no whitespaces)
;;             * each section has to be ended by a line only con-
taining "}"
;;
;;
;; All functions and groups need to be defined as simple tags
INSIDE the section "FunctionList".
;; Function can be grouped (especially for printing grouped re-
sults to Spreadsheet-Files)
;; by the following syntax
;; { group = MyGroup1
;; function1
;; function2
;; function3
;; function4
;; }

;; In the section "Global" various settings can be given to con-
trol memtrace.
;; Currently only BaseAddr and PageSize of data memory can be ap-
plied for page hit/miss calculation
;; You can un-comment the following lines in order to apply these
settings:

; { Global = Global
;   BaseAddr = 0x0  ; Base address must be expressed as
;                     hexadecimal number (default is 0x0)
;   PageSize = 128  ; Page Size must be represented as
;                     integer value > 0 (default is 128)
; }
```

```
{ Global = Global
   BaseAddr                  = 0x0
   PageSize                  = 128
   StackBaseAddr             = 0x80000000
   StackSize                 = 8192
   AdditionalArmsdParameters = ""
   AxfFileParameters         = ""
   BusDivisor                = 2
   EnableTracer              = 0
   ProcessorType             = "ARM946E-S"
   AxfFileName               = "D:\h264enc\bin\arm\testappl.axf"
   AnalysisResultFileName    = "D:\h264enc\memtrace_out_Func.txt"
   XlsFileName               = "D:\h264enc\memtrace_out_Funct.xls"
   ObjectFile                = "D:\h264enc\objects\*.o"
   TraceType                 = 0
   TableType                 = 2
   TableTypeVar              = ""
   RangeBegin                = 0
   RangeEnd                  = 0
   RangeStep                 = 0
   RangePeriod               = 0
   RangePeriodEnd            = 0
   EnablePeriod              = 0
   EnableRange               = 0
   ClockSpeed                = 200
   ClockSpeedUnit            = M
   Memory = "0 100000000 DRAM 4 rw 230/10 230/10"
}

{ FunctionList = FunctionList
      ; * for creating intermediate results set the
      ;   value of a function to "split" (functionName = split)
      ; * for weighting the cycle count results of a
      ;   function when creating spreadsheet tables
      ;   set the value of a function to the weighting factor
      ;   with a leading "*" (functionName = *0.4578

  CalcSNRFrame     = ":D:\h264enc\objects\testvidenc.r.o"
  CloseYUVFile     = ":D:\h264enc\objects\testvidenc.r.o"
  OpenYUVFile      = ":D:\h264enc\objects\testvidenc.r.o"
  ReadYUVFrame     = ":D:\h264enc\objects\testvidenc.r.o"
  WriteRecFrame    = ":D:\h264enc\objects\testvidenc.r.o"
  main             = ":D:\h264enc\objects\testvidenc.r.o"
…
{ group = BitStreamEncoding
  writeCoeffBlockCAVLC       = ":D:\h264enc\objects\cavlc.r.o"
  writeLevelVLC0             = ":D:\h264enc\objects\cavlc.r.o"
  writeLevelVLCN             = ":D:\h264enc\objects\cavlc.r.o"
  writeNumCoeffTrailingOnes  = ":D:\h264enc\objects\cavlc.r.o"
  writeNumCoeffTrailingOnesChromaDC
                             =  ":D:\h264enc\objects\cavlc.r.o"
  writeRun                   = ":D:\h264enc\objects\cavlc.r.o"
  PutBits                    = ":D:\h264enc\objects\bitstream.r.o"
  putCoeffChromaBlock        = ":D:\h264enc\objects\scancoeff.r.o"
  putCoeffChromaBlockDC      = ":D:\h264enc\objects\scancoeff.r.o"
```

```
  putCoeffLumaBlockAC        = ":D:\h264enc\objects\scancoeff.r.o"
  putCoeffLumaBlockDC        = ":D:\h264enc\objects\scancoeff.r.o"
}
{ group = ChromaPrediction
  TransQuantChroma8  = ":D:\h264enc\objects\transquantchroma8.r.o"
  enc_predictChroma         = ":D:\h264enc\objects\predict.r.o"
  IntraPredChroma_noRD      = ":D:\h264enc\objects\intrapred8.r.o"
}
}

{ MemoryMap = MemoryMap

  { FixedAreaList = FixedAreaList
  ; add abitrary memory regions here
  ; format: <name> = <startaddress> <size>
  ; (startaddress and size can be
  ;  decimal, hex (leading '0x') or octal (leading '0')
  ; e.g.: stack = 0x4E001 8192

  }

  { SectionList = SectionList
  'ER_RO'
  'ER_RW'
  'ER_ZI'
  }

  { GlobalVariableList = GlobalVariableList
  yuvfile
  yuvfile
  tz_chromadc_lentab
  tz_chromadc_codtab
  totalzeros_lentab
  enc_predictIntra4_table
  dequantcoef
  dequant_coef
  clip_zero
  clip_lut
  block_intra4x4mode_slicetable
  block_indexes
  QP_SCALE_CR
  NumCoeffTrailingOnes_Lengths_3
  NumCoeffTrailingOnes_Lengths_2
  NumCoeffTrailingOnes_Lengths_1
...
  NCBP_ENC
  COEFF_COSTC
  }
}
```

*Listing 15: MEMTRACE configuration file*

### 8.2.5 List of Source Code Files

Table 41 shows a list of the source code files of the MEMTRACE profiling suite, including the common frontend parts of GUI and CLI and the backend of the tool.

*Table 41: Source code files*

| File | Purpose |
|---|---|
| **./Src:** | |
| **Getopt.c** | Functions for command-line parameter reading |
| **Getopt1.c** | Additional functions for long command-line parameters |
| **Memtrace.cpp** | Main program for executable |
| **Initmode.cpp** | Functions and classes for init mode in memtrace.cpp |
| **Runmode.cpp** | Functions and classes for run mode in memtrace.cpp |
| **Spreadsheetmode.cpp** | Functions and classes for spreadsheet mode in memtrace.cpp |
| **Memtrace_common.cpp** | Shared functions and classes of executable and dll |
| **Memtrace_dll.cpp** | Functions and classes for dll |
| **Tracer_for_memtrace_dll.c** | Modified tracer.c, which acts as interface between ARMulator and the memtrace dll functions |
| **Copro_basics.c** | Functions for coprocessor template |
| **Bus_controller.cpp** | Functions for memory bus profiling and DMA controller |
| **Mapfile_for_memtrace.c** | Modified mapfile.c, which includes calls to bus controller functions |
| **Sordi.def** | Definition file required for the creation of an ARMulator DLL |
| **./Include:** | |
| **Getopt.h** | Header file |
| **Memtrace_common.h** | Header file |
| **Memtrace_dll.h** | Header file |
| **Memtrace_global.h** | Header file |
| **Initmode.h** | Header file |
| **Runmode.h** | Header file |
| **Spreadsheetmode.h** | Header file |
| **Copro_basics.h** | Header file |
| **Mapfile.h** | Header file |
| **./** | |
| **Memtrace_dll.dsp** | Project file for dll |
| **Memtrace_copro.dsp** | Project file for coprocessor template |
| **Memtrace_mapfile.dsp** | Project file for extended bus model |
| **Memtrace.dsp** | Project file for executable |
| **Memtrace.dsw** | Workspace file, including dll and exe projects |

### 8.2.6 Full Description of the Command-line Syntax

```
memtrace [-i] [-r] [-x] [-y] [-c configuration-file]
         [-a executable-file]  [-p exe-file-parameters]
         [-m output-file]
         [-e spreadsheet-file]  [-f  spreadsheet-output-format]
         [-d debugger-options] [-t][-?] [-V]  [-o object-files]
```

| | |
|---|---|
| `-i` | Init mode: creates memtrace.ini file with list of local functions from the object-files |
| `-r` | Run mode: starts the profiling run in conjunction with the ISS |
| `-x` | Postprocessing mode: creates tab-separated spreadsheet file from MEMTRACE function profiling output |
| `-y` | Postprocessing mode: Creates tab-separated spreadsheet file from MEMTRACE variable profiling output |
| `-c configuration-file` | Defines the name of the configuration file as either created in init mode or used in run and spreadsheet mode. |
| `-o object-files` | Defines the object files, libraries and archives. Multiple space-separated paths/files can be supplied. IMPORTANT: It must be the last option on the command-line |
| `-d debugger-options` | Defines options to be passed over to the debugger/ISS. Useful options are for example setting the processor type, speed or cache parameters. The availability of the options is dependent on the debugger/ISS. When using the ARMulator, see armsd help ("armsd -h") for more options. IMPORTANT: enclose debugger options in " " |
| `-a executable-file` | Specifies executable (axf-) input file for profiling |
| `-p exe-file-parameters` | Specifies command-line parameters for the executable |
| `-m output-file` | Defines the name of the MEMTRACE output file. Default output file is memtrace_out.txt. |
| `-e spreadsheet-output-file` | Defines the name of the tab-separated spreadsheet output file. Default spreadsheet output file: memtrace_out.xls. |
| `-f spreadsheet-output-format` | Output format of the spreadsheet file, see Section 5.4. The default format is “{ov}” creating a table with overall results. |
| `-t` | Turn on tracer module for tracing instructions, memory accesses and events and writes them to a file. The availability is dependent on the ISS. For the ARMulator, see Section 2.2.1.1. |
| `-? or -h` | Display usage information and exit. |
| `-V` | Display version number of MEMTRACE. |

# References


[1] Altera, *Hardware Reference Manual: Excalibur - ARM-Based Embedded Processor PLDs*, 2002.

[2] G.M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", *Proceedings of the AFIPS Spring Joint Computer Conference*, 1967, pp. 483-485.

[3] Analog Devices Inc., *ADSP-BF531/ADSP-BF532/ADSP-BF533: Blackfin Embedded Processor Data Sheet*, 2007.

[4] Analog Devices Inc., *AD623 - Single Supply, Rail-to-Rail, Low Cost Instrumentation Amplifier*, 1999.

[5] Analog Devices Inc., *ADSP-BF533 Blackfin Processor Hardware Reference*, 2003.

[6] ARC International, *ARC Website*, http://www.arc.com.

[7] ARC International, *Integrated Profiler User's Guide*, 2004.

[8] ARM Ltd., *ARM Website*, http://www.arm.com.

[9] ARM Ltd., *Application Note 26: Benchmarking, Performance Analysis and Profiling*, 1995.

[10] ARM Ltd., *AMBA Specification (Rev 2.0)*, 1999.

[11] ARM Ltd., *AMBA AXI Protocol v1.0*, 2004.

[12] ARM Ltd., *Writing Efficient C for ARM (DAI 0034A)*, 1998.

[13] ARM Ltd., *RealView Compilation Tools Version 2.1: Compiler and Libraries Guide (DUI 0205D)*, 2004.

[14] ARM Ltd., *RealView Developer Suite Version 2.1: Getting Started Guide (DUI 0255B)*, 2004.

[15] ARM Ltd., *RealView ARMulator ISS Version 1.4 User Guide (DUI 0207C)*, 2004.

[16] ARM Ltd., "ARM holdings PLC reports second quarter and half year 2006 results", http://www.arm.com/news/14087.html, 2006.

[17] T. Austin, T. Mudge and D. Grunwald, "PowerAnalyzer for pocket computers", http://www.eecs.umich.edu/lpanalyzer/pdfs/contractorsFall01.pdf, 2001.

[18] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan and P. Marwedel, "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems", *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES)*, 2002, pp. 73-78.

[19] Bell Laboratories, *Unix Programmer's Manual, Section 1, "Prof Command"*, 1979.

[20] J.P. Bentley, *Principles of Measurement Systems*, 4th edition, Pearson Prentice Hall, Upper Saddle River, NJ, 2005.

[21] Berkeley Design Technology, Inc. (BDTI), "Embedded system", *BDTI's DSP Dictionary*, http://www.bdti.com/articles/dspdictionary.html.

[22] H. Blume, H. Hübert, H. T. Feldkämper and T. G. Noll, “Model-based exploration of the design space for heterogeneous systems on chip”, *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, 2002, pp. 29-40.

[23] H. Blume, D. Becker, L. Rotenberg, M. Botteck, J. Brakensiek and T. G. Noll, “Hybrid functional- and instruction-level power modeling for embedded and heterogeneous processor architectures”, *Journal of Systems Architecture*, vol. 53, no. 10, pp. 689-702, Oct. 2007.

[24] H. Blume, J. von Livonius, L. Rotenberg, T. G. Noll, H. Bothe and J. Brakensiek, “OpenMP-based parallelization on an MPCore multiprocessor platform - A performance and power analysis”, *Journal of Systems Architecture*, vol. 54, no. 11, pp. 1019-1029, Nov. 2008.

[25] J. Bormans, K. Denolf, S. Wuytack, L. Nachtergaele and I. Bolsens, “Integrating system-level low power methodologies into a real-life design flow”, *Proceedings of the the Ninth International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 1999, pp. 19-28.

[26] D. Brooks, V. Tiwari and M. Martonosi, “Wattch: A framework for architectural-level power analysis and optimizations”, *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000, pp. 83 - 94.

[27] D. Burger and T. M. Austin, “The SimpleScalar tool set, version 2.0”, *ACM SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13-25, Jun. 1997.

[28] N. Chang and K. Kim, “Real-time per-cycle energy consumption measurement of digital systems”, *IEE Electronics Letters*, vol. 36, no. 13, pp. 1169-1171, Jun. 2000.

[29] CoWare Inc., “Processor designer“, http://www.coware.com/products/processordesigner.php.

[30] CoWare Inc., *LISATek Processor Debugger Manual*, 2006.

[31] J. Edler and M. D. Hill, “Dinero IV : Trace-driven uniprocessor cache simulator”, http://pages.cs.wisc.edu/~markhill/DineroIV/.

[32] S. Edwards, L. Lavagno, E.A. Lee and A. Sangiovanni-Vincentelli, “Design of embedded systems: formal models, validation, and synthesis”, *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366-390, Mar. 1997.

[33] European Telecommunications Standards Institute (ETSI), *Digital video broadcasting (DVB); Transmission System for Handheld Terminals (DVB-H), ETSI EN 302 304 V1.1.1 (2004-11)*, 2004.

[34] M.J. Flynn, “Some computer organizations and their effectiveness”, *IEEE Transactions on Computers*, vol. 21, no. 9, pp. 948-960, Sept. 1972.

[35] Free Software Foundation, “GNU binutils”, http://www.gnu.org/software/binutils/.

[36] Free Software Foundation, “GCC, the GNU Compiler Collection”, http://gcc.gnu.org/.

[37] S. Furber, *ARM System-on-Chip Architecture*, 2nd edition, Addison-Wesley Longman, Amsterdam, The Netherlands, 2000.

[38] J. Gaisler, “A portable and fault-tolerant microprocessor based on the SPARC v8 architecture”, *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2002, pp. 409-415.

[39] D.D. Gajski and F. Vahid, “Specification and design of embedded hardware-software systems”, *IEEE Design & Test of Computers*, vol. 12, no. 1, pp. 53-67, 1995.

[40] R. Goering, “Startup analyzes algorithms for power consumption”, *EE Times*, http://www.eetimes.com/news/design/showArticle.jhtml?articleID=17500869, 2004.

[41] R.E. Gonzalez, “Xtensa: a configurable and extensible processor”, *IEEE Micro*, vol. 20, no. 2, pp. 60-70, Mar.-Apr. 2000.

[42] S.L. Graham, P.B. Kessler and M.K. McKusick, “Gprof: A call graph execution profiler”, *Proceedings of the SIGPLAN Symposium on Compiler Construction*, 1982, pp. 120-126.

[43] P. Guerrier and A. Greiner, “A generic architecture for on-chip packet-switched interconnections”, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2000, pp. 250-256.

[44] S. Ha, C. Lee, Y. Yi, S. Kwon and Y.-P. Joo, “Hardware-software codesign of multimedia embedded systems: The PeaCE”, *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2006, pp. 207-214.

[45] J. L. Hennessy and D. A. Patterson, *Computer Architecture : A Quantitative Approach*, 4th edition, Morgan Kaufmann Publishers, San Francisco, CA, 2006.

[46] H. Hübert, B. Stabernack and H. Richter, “Tool-aided performance analysis and optimization of an H.264 decoder for embedded systems”, *Proceedings of the Second IEEE International Symposium on Consumer Electronics (ISCE)*, 2004, pp. 400-405.

[47] H. Hübert, B. Stabernack and H. Richter, “Tool-aided performance analysis and optimization of multimedia applications”, *Proceedings of the Second Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia)*, 2004, pp. 99-104.

[48] H. Hübert, B. Stabernack and K.-I. Wels, “Performance and memory profiling for embedded system design”, *Proceedings of the 10th International Symposium on Industrial Embedded Systems (SIES)*, 2007, pp. 94-101.

[49] H. Hübert and B. Stabernack, “Power modeling of an embedded RISC core for function-accurate energy profiling”, *Proceedings of the 12th Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systeme*, 2009, pp. 147-156.

[50] H. Hübert, “A survey of HW/SW cosimulation techniques and tools”, Stockholm, Sweden, Thesis work, Royal Institute of Technology (KTH), 1998.

[51] IBM Inc., *Whitepaper: CoreConnect™ Bus Architecture*, 1999.

[52] Infineon Technologies, *Architecture Overview Handbook: TriCore 1.3 - 32-bit Unified Processor Core*, 2002.

[53] Intel Inc., *Application Note: StrongARM SA-110 Microprocessor Instruction Timing*, 1998.

[54] Intel Inc., *PC SDRAM Specification (Revison 1.7)*, 1999.

[55] Intel Inc., *3 Volt Intel StrataFlash Memory 28F128J3A, 28F640J3A, 28F320J3A (x8/x16)*, 2001.

[56] Intel Inc., *Intel PXA27x Processor Family Developer's Manual*, 2004.

[57] Intel Inc., "Intel VTune performance analyzers", http://www.intel.com/software/products/vtune/.

[58] S.S. Iyer and H.L. Kalter, "Embedded DRAM technology: opportunities and challenges", *IEEE Spectrum*, vol. 36, no. 4, pp. 56-64, Apr. 1999.

[59] J. Janzen, *Calculating Memory System Power for DDR SDRAM [J/OL] . Micron Design Line*, Micron Technology Inc., 2001.

[60] Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T, VCEG, *International Standard of Joint Video Specification (ITU-T Rec. H.264— ISO/IEC 14496-10 AVC), JVT-G050*, Mar. 2003.

[61] N. Julien, J. Laurent, E. Senn and E. Martin, "Power Consumption Modeling and Characterization of the TI C6201", *IEEE Micro*, vol. 23, no. 5, pp. 40-49, Sept. 2003.

[62] P. M. Kuhn and W. Stechele, "Complexity analysis of the emerging MPEG-4 standard as a basis for VLSI implementation", *Proceedings of the SPIE Visual Communications and Image Processing (VCIP)*, 1998, pp. 498-509.

[63] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja and A. Hemani, "A network on chip architecture and design methodology", *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 2002, pp. 105-112.

[64] J. Levon and P. Elie, "Oprofile: A system profiler for Linux", http://oprofile.sourceforge.net/, 2005.

[65] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, revised edition, John Wiley & Sons Inc, Hoboken, NJ, 1990.

[66] P. Marwedel, *Embedded System Design*, Springer, Dordrecht, The Netherlands, 2003.

[67] Maxim Integrated Products Inc., *Maxim 4376 - Single/Dual/Quad, High-Side Current-Sense Amplifiers with Internal Gain*, 2004.

[68] Mazdak & Alborz Design Automation, "Reusing Verilog IP cores in SystemC Environment by V2SC", http://www.mazdak-alborz.com/download/v2sc-IPSOC2005.pdf, 2005.

[69] Mentor Graphics Corporation, *ModelSim User's Manual Software Version 6.2g*, 2007.

[70] V.G. Moshnyaga and H. Tsuji, "Cache energy reduction by dual voltage supply", *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2001, pp. 922-925.

[71] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation", *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 89-100.

[72] Netrino and The Embedded Systems Experts, "Embedded system", *Embedded Systems Glossary*, http://www.netrino.com/Embedded-Systems/Glossary-E.

[73] OpenCores Project, "Wishbone version B3", http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf.

[74] P. R. Panda, N.D. Dutt, A. Nicolau, F. Catthoor, A. Vandecappelle, E. Brockmeyer, C. Kulkarni and E. De Greef, “Data memory organization and optimizations in application-specific systems”, *IEEE Design & Test of Computers*, vol. 18, no. 3, pp. 56-68, May-Jun. 2001.

[75] P. P. Pande, C. Grecu, M. Jones, A. Ivanov and R. Saleh, “Performance evaluation and design trade-offs for network-on-chip interconnect architectures”, *IEEE Transactions on Computers*, vol. 54, no. 8, pp. 1025-1040, Aug. 2005.

[76] S. Pees, A. Hoffmann, V. Zivojnovic and H. Meyr, “LISA - machine description language for cycle-accurate models of programmable DSP architectures”, *Proceedings of the 36th ACM/IEEE Design Automation Conference (DAC)* , 1999, pp. 933-938.

[77] G. Qu, N. Kawabe, K. Usami and M. Potkonjak, “Function-level power estimation methodology for microprocessors”, *Proceedings of the 37th ACM/IEEE Design Automation Conference (DAC)*, 2000, pp. 810-813.

[78] J.A. Rowson, “Hardware/software co-simulation”, *Proceedings of the 31st ACM/IEEE Design Automation Conference (DAC)*, 1994, pp. 439-440.

[79] Samsung Electronics, *128Mbit SDRAM 2M x 16Bit x 4 Banks Synchronous DRAM LVTTL (K4S281632C CMOS SDRAM)*, 2000.

[80] A. Scandurra, G. Falconeri and B. Jego, *STBus Communication System: Concepts and Definitions*, 2002.

[81] G. Schirner, G. Sachdeva, A. Gerstlauer and R. Dömer, “Modeling, simulation and synthesis in an embedded software design flow for an ARM processor” , Technical Report 06-06, Center for Embedded Computer Systems, University of California Irvine, 2006.

[82] O. Schreer and S. Ngongang, “Real-time gesture recognition in advanced videocommunication services”, *Proceedings of the 14th International Conference on Image Analysis and Processing (ICIAP)*, 2007, 253-258.

[83] T. Simunic, L. Benini and G. De Micheli, “Cycle-accurate simulation of energy consumption in embedded systems”, *Proceedings of the 36th ACM/IEEE Design Automation Conference (DAC)*, 1999, pp.867-872.

[84] A. Sinha and A. P. Chandrakasan, “JouleTrack: A web based tool for software energy profiling”, *Proceedings of the 38th ACM/IEEE Design Automation Conference (DAC)*, 2001, pp. 220-225.

[85] W. Snyder, P. Wasson and D. Galbi, “Introduction to Verilator”, http://www.veripool.com/verilator.html, 2008.

[86] B. Stabernack, H. Hübert and K.-I. Wels, “A Companion chip for H.264/AVC video Processing”, *Proceedings of the Global Signal Processing Conference and Expo (GSPx)*, Oct. 2006.

[87] B. Stabernack, H. Hübert and K.-I. Wels, “Terminal architectures for DVB-H”, *Proceedings of the Global Signal Processing Conference and Expo (GSPx-TV to Mobile)*, Mar. 2006.

[88] B. Stabernack, H. Hübert and K.-I. Wels, “A H.264 video coprocessor for mobile DVB-H terminals”, *Proceedings of the IEEE International Conference on Consumer Electronics(ICCE)*, 2006, pp. 89-90.

[89] B. Stabernack, K.-I. Wels and H. Hübert, “A video coprocessor for mobile multi media signal processing”, *Proceedings of the 11th IEEE International Symposium on Consumer Electronics (ISCE)*, 2007, 1-6.

[90] B. Stabernack, “Architekturkonzepte für prozessorbasierte MPEG Videodecoder mit Schwerpunkt für mobile Anwendungen“, Ph. D. thesis, Technical University Berlin, Germany, 2004.

[91] B. Stabernack, K.-I. Wels and H. Hübert, “A system on a chip architecture of an H.264/AVC coprocessor for DVB-H and DMB applications”, *IEEE Transactions on Consumer Electronics*, vol. 53, no. 4, pp. 1529-1536, Nov. 2007.

[92] B. Stabernack, K.-I. Wels and H. Hübert, “Hardware and software architectures for mobile multimedia signal processing”, B. Furht (editor) and S. A. Ahson (editor), *Handbook of Mobile Broadcasting: DVB-H, DMB, ISDB-T, AND MEDIAFLO*, Auerbach Publications Inc., Boca Raton, FL, pp. 95-132, 2008.

[93] R.A. Sugumar and S.G. Abraham, “Efficient simulation of caches under optimal replacement with applications to miss characterization”, *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1993, pp. 24-35.

[94] Tensilica Inc., *Tensilica Website*, http://www.tensilica.com/.

[95] Texas Instruments Inc., *PT6980: 10-A 12V-Input Dual Output Series Integrated Switching Regulator*, 2001.

[96] S. Thoziyoor, N. Muralimanohar and N. P. Jouppi, “CACTI 5.0”Technical Report HPL-2007-167, Advanced Architecture Laboratory, HP Laboratories, Palo Alto, CA, 2007.

[97] V. Tiwari, S. Malik, A. Wolfe and M. T.-C. Lee, “Instruction level power analysis and optimization of software”, *The Journal of VLSI Signal Processing*, vol. 13, no. 2-3, pp. 1-18, Aug. 1996.

[98] Department of Computer Engineering, University of Tübingen, “VHDL-to-SystemC-converter”, *European SystemC Users Group*, http://www-ti.informatik.uni-tuebingen.de/~systemc/.

[99] L. Wehmeyer and P. Marwedel, *Fast, Efficient and Predictable Memory Accesses: Optimization Algorithms for Memory Architecture Aware Compilation*, Springer, Dordrecht, The Netherlands, 2006.

[100] W.H. Wolf, “Hardware-software co-design of embedded systems”, *Proceedings of the IEEE*, vol. 82, no. 7, 967-989, Jul. 1994.

[101] D. Wright and B. Freeman-Benson, “How to write an eclipse debugger”, http://www.eclipse.org/articles/Article-Debugger/how-to.html, Aug. 2004.

[102] W. Ye, N. Vijaykrishnan, M. Kandemir and M.J. Irwin, “The design and use of SimplePower: a cycle-accurate energy estimation tool”, *Proceedings of the 37th ACM/IEEE Design Automation Conference (DAC)*, 2000, pp. 340-345.

[103] K.-S. Yeo and K. Roy, *Low Voltage, Low Power VLSI Subsystems*, McGraw-Hill Education, New York, NY, 2004.