Enhancing the Scalability of Many-core Systems
Towards Utilizing Fine-Grain Parallelism in Task-Based Programming Models

vorgelegt von
M.Sc.
Tamer Dallou
geb. in Irbid - Jordanien

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

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Stephan Kreutzer
Gutachter:	Prof. Dr. Ben Juurlink
Gutachter:	Prof. Dr. Jesus Labarta
Gutachter:	Prof. Dr. Jeronimo Castrillon

Tag der wissenschaftlichen Aussprache: 13. Dezember 2017

Berlin 2017

Abstract

In the past few years, it has been foreseeable that Moore's law is coming to an end. This law, based on the observation that the number of transistors in an integrated chip doubles every 18-24 months, served as a roadmap for the semiconductors industry. On the verge of its end due to the huge increase in integrated chips power density, a new era in computing systems has begun. In this era, a core's single performance is no longer the most important parameter, but the performance of the whole multicore system. It is an era where multiplicity and heterogeneity of computing units became the norm in state-of-the-art systems on chips (SoCs), not the exception.

Although the problem of programmability of such complex systems has been addressed in both academia and industry, the pace at which more integration of processing cores on chip was faster than that of bringing user applications to adapt and scale well on these chips. New programming models emerged trying to bridge the gap between programming complexity and well-utilization of the multicore systems, with their various available resources.

One promising approach is the dataflow task-based programming model, where an application is broken down to smaller execution units called tasks, which will dynamically be scheduled to run on the available resources according to data dependences between those tasks. However, this approach has an overhead as its runtime system needs considerable amount of computational power to track dependences between tasks and build the task graph, decide on ready tasks, schedule them on idle cores, and upon task completion, kick off dependent tasks, all this performed dynamically at runtime. Although dataflow task-based programming provides a solution to the programmability problem of multicore systems, its runtime overhead, in practice, has limited the scalability of applications programmed using this programming model, especially when the tasks are fine-grain, and/or have complex task graphs.

In case of applications that could be broken down to coarse-grain tasks, the runtime overhead becomes less relevant, as the worker cores stay busy executing the coarse-grain tasks long enough for the runtime system to simultaneously maintain the task graph and find the next batch of ready tasks to feed the worker cores once they ask for more work.

However, having control on the granularity of tasks is not a trivial task, as this requires grouping tasks together in order to form coarser tasks. The grouping of tasks implies knowledge of the dependences between them in order to reflect them on larger tasks so that the behavior of the application remains unaltered. Although this work-around to hide the runtime overhead pushes the dataflow programming model further as a promising solution for utilizing multicore systems, it (the grouping of tasks) is usually done by the programmer, which again raises the complexity of programming.

Moreover, grouping tasks becomes more complex for applications that can be broken down to tens, if not hundreds or thousands of (fine-grain) tasks. As the number and heterogeneity of cores is increasing, efficiently mapping a huge number of inter-dependent tasks on modern multicore systems is simply beyond the human capabilities.

For the above reasons, a logical approach is to skip the grouping of tasks, and offload the task graph management to another entity other than the runtime system: a hardware accelerator for example.

About 15 years ago, at the time of single core processors of relatively low gate count which were manufactured using the 180 nm technology, the overhead of a dedicated hardware accelerator was significant when considering the extra power consumption, overall chip area, and the resulting heat. In today's measures this is not the case anymore, as the semiconductor technology has advanced dramatically and multicore processors are now the standard. Given the enormous number of transistors on chip and the complexity of the individual processors, adding a dedicated hardware accelerator became justifiable.

The main contribution of this thesis is to offload the heaviest part of the runtime system - task graph management - to a dedicated hardware accelerator, in order to accelerate the runtime system, as well as to save some conflicts on using the shared resources (microprocessor cores and memory system) by the runtime system and the user applications. Moreover, a high-level application programming interface is pre-

sented which enables system programmers of utilizing the proposed hardware task graph manager to accelerate their own runtime systems.

In this thesis, the programmability problem of multicore systems has been analyzed, highlighting the StarSs/OmpSs models as representative candidates of dataflow task-based programming models and the need to make use of the very fine-grain tasks. Afterwards, hardware support for such programming models is presented in the form of the Nexus++ co-processor for the StarSs/OmpSs programming models. Using traces of applications written in StarSs/OmpSs from the StarBench benchmark suite, Nexus++ significantly improves the scalability of those applications.

To demonstrate its functionality in real applications, Nexus++ has been implemented on an FPGA board and integrated with VSs: a light-weight runtime system that implements the part of OmpSs runtime system that is needed to run most of the StarBench benchmarks. Plugging the FPGA board into any multicore machine PCIe slot and installing the necessary tools, Nexus++ can successfully manage applications with tens of thousands of tasks.

Nexus# is the successor of Nexus++, which has an improved execution pipeline compared to Nexus++, in addition to parallelizing the process of task graph management itself in a distributed fashion. For example, running an application with coarse-grain independent tasks such as *ray tracing* and where a software-only parallel solution achieves 31× speedup compared to the serial execution, Nexus# and Nexus++ achieve speedups of 194× and 60× respectively.

Nexus# outperforms Nexus++ in terms of scalability, which opens the door to support even finer-grain tasks. For the case of fine-grain tasks with complex inter-task dependencies as in H.264 video decoding, the software-only parallel solution is slower than the serial execution due to runtime overhead, but Nexus# and Nexus++ achieve speedups of 7× and 2.2× respectively.

As described in this work, through its extensive reconfigurability, Nexus# presents a suitable hardware accelerator for various multicore systems, ranging from embedded to complex high-performance systems.

Zusammenfassung

Seit einigen Jahren ist absehbar, dass die Gültigkeit von Moore's Law nach fast 50 Jahren zu Ende geht. Dieses "Gesetz" basierte auf der Beobachtung, dass sich die Anzahl der Bauelemente auf einem Chip etwa alle 18 Monate verdoppelte und legte den Grundstein für die Roadmap der Halbleiterindustrie.

Aufgrund der enormen Zunahme der Leistungsdichte von Mikroprozessoren steht diese Gesetz kurz vor dem Ende und ein neues Zeitalter in der Computerindustrie bricht an. In diesem Zeitalter ist die Single-Thread-Performance kein so wichtiger Faktor mehr, im Vergleich zur Leistung des gesamten Multicore-Systems. Es ist ein Zeitalter in dem die Vielfalt und Vielzahl von Prozessoren die Norm in aktuellen System on Chips (Soc) ist und nicht die Ausnahme.

Obwohl das Problem der Programmierbarkeit solcher komplexen Systeme sowohl von der Forschung als auch der Industrie angegangen wurde, hat die Entwicklung auf dem Softwaresektor nicht mit der Entwicklung der Hardware Schritt gehalten. Seitdem sind neue Programmiermodelle entstanden, die versuchen, die Lücke zwischen algorithmischer Komplexität und einer effizienten Nutzung aller Ressourcen der SoC-Hardware zu überbrücken.

Ein vielversprechender Ansatz ist das Datenfluss taskparallele Modell. In diesem wird das Anwendungsprogramm in kleine Arbeitseinheiten, engl. Task, zerlegt. Diese Tasks werden dann unter Berücksichtigung der Datenabhängigkeiten dynamisch auf geeignete und verfügbare Ressourcen verteilt und parallel abgearbeitet.

Dabei muss zwischen verschiedenen Laufzeitsystemen und Varianten dieses Programmiermodells abgewogen werden. Bei der Auswahl müssen folgende Punkte berücksichtigt werden, da diese eine nicht unerhebliche Rechenleistung benötigen: die interne Verwaltung des Taskgraphen, das Verfolgen der Datenabhängigkeiten zwischen Tasks, die Entscheidung wann ein Task bereit zur Bearbeitung ist, die Verteilung auf oder Zuteilung von Prozessoren und schließlich, nach der Abarbeitung der Aufgabe,

die Rückmeldung an das Laufzeitsystem, die die Freigabe von weiteren Tasks nach sich zieht.

Für Datenfluss taskparallele Programmiermodelle, in vielen Fällen wenigstens theoretisch eine sinnvolle Möglichkeit zur Programmierung von Multi- und Many-core Systemen, stellt der Overhead im Laufzeitsystem in der Praxis vielfach eine unüberwindbare Hürde dar. Dies gilt insbesondere für sehr fein unterteilte oder sehr komplexe Datenfluss- oder Taskgraphen.

Für Anwendungsprogramme, die sich in große Tasks zerlegen lassen, ist der Aufwand im Laufzeitsystem weniger relevant, da die Kosten dafür im Verhältnis zur geleisteten Arbeit amortisiert werden. Das Laufzeitsystem kann hier mit relativ geringem Aufwand gleichzeitig den Datenflussgraphen verwalten und Arbeit an die Hardwareressourcen verteilen, um diese damit effektiv auszulasten.

Es ist jedoch keine triviale Aufgabe, die Granularität durch das Zusammenfassen kleinerer Tasks zu größeren Gruppen zu verbessern. Dies setzt genaues Wissen über die Struktur des Datenflusses zwischen den einzelnen Tasks voraus, da die Semantik der Applikation natürlich nicht verändert werden darf. Trotz des Workarounds zur Verschleierung des Laufzeitmehraufwandes, stellen die taskbasierten Programmiermodelle eine vielversprechende Lösung zur Auslastung von Multicore-Systemen dar. Diese Tätigkeit wurde bisher von den Programmierern erledigt, was eine höhere Komplexität von Programmen für Multicore-Systemen nach sich zog.

Die Komplexität der Aufgabe wächst zukünftig mit der Anzahl und der Vielfalt der Prozessorkerne, sowie mit Applikationen, die nicht in zehn, sondern in hunderte oder gar tausende von kleinen Fragmenten zerteilt werden. Diese können dann potentiell parallel ausgeführt werden. Die Fragmente zu gruppieren und an das Laufzeitsystem zur Ausführung zu übergeben, übertrifft die Leistungsfähigkeit eines normalen Menschen.

Aus den genannten Gründen ist es sinnvoll, die Gruppierung von Tasks zu vermeiden, und stattdessen Funktionen wie zum Beispiel die Graph-Verwaltung, in einen Coprozessor zu verlagern, um das Laufzeitsystem zu entlasten.

Vor etwa 15 Jahren waren Single-Core Prozessoren der Stand der Technik. In der damals verfügbaren 180 nm Technologie war der zusätzliche Aufwand an Gattern für einen Beschleuniger wirtschaftlich nicht zu rechtfertigen. Die meisten Computersys-

teme verfügten ohnehin nur über eine CPU und Programme bestanden auch nur aus einem Thread.

Nach heutigem Maßstab ist dies nicht mehr der Fall, die Halbleitertechnologie hat sich erheblich weiterentwickelt. Multiprozessorsysteme sind der Stand der Technik und gemessen an der Zahl der verfügbaren Gatter, sowie der Komplexität einzelner CPUs, ist der Aufwand für einen HW Beschleuniger vernachlässigbar gering.

Der Hauptbeitrag dieser Dissertation ist es, den größten Teil eines Laufzeitsystems, das Task-Graph-Management, auf einen dedizierten Hardwarebeschleuniger auszulagern. Ziel dabei ist es, die Laufzeit des Systems zu steigern und gleichzeitig einigen Konflikten vorzubeugen, die entstehen, wenn das Laufzeitsystem oder Anwenderapplikationen geteilte Ressourcen (Mikroprozessorkerne und Speichersysteme) verwenden.

Zusätzlich wird eine high-level Programmierschnittstelle präsentiert, welche es dem Systemprogrammierer ermöglicht den vorgeschlagenen Hardware-Taskgraph-Manager auszulasten, um ihr eigenes Laufzeitsystem zu beschleunigen.

In dieser Arbeit wurde die Programmierbarkeit von heterogenen Multiprozessorsystemen mit Schwerpunkt auf taskparallele Programmiermodelle analysiert, sowie der Nachteil von gelegentlich sehr kleinen Arbeitseinheiten (Tasks). Danach wird die Hardware-Unterstützung für das StarSs/OmpS Programmiermodell in Form des Nexus ++ Co-Prozessors eingeführt und der Einfluss auf Skalierbarkeit und Systemleistung gemessen.

Für Traces von Anwendungen aus der StarBench Benchmarksuite zeigte Nexus++ eine erheblich verbesserte Skalierbarkeit dieser in StarSs/OmpSs geschriebenen Programme.

Um die Funktionalität im echten Anwendungsfall und die Leistungsfähigkeit des Konzepts zu beweisen, haben wir Nexus++ auf einem FPGA Board implementiert. Die Kommunikation mit dem Host-System erfolgte über eine PCIe Schnittstelle. Auf der Softwareseite wurde Nexus++ in VSs, ein light-weight Laufzeitsystem, integriert. VSs unterstützt die wesentlichen Funktionen des OmpSs Laufzeitsystems und erlaubt es die meisten Programme der StarBench Benchmarksuite auszuführen.

Durch den Einsatz des FPGA Boards als Beschleuniger in den PCIe Slot eines beliebigen Mehrkernsystems und Installation der notwendigen Softwarewerkzeuge, konnte

gezeigt werden, dass Nexus++ in der Lage ist, Applikationen mit mehreren zehntausend Tasks zu verwalten.

Nexus# ist eine überarbeitete Version von Nexus++. Zusätzlich zur Parallelisierung der Prozesse für die Task-Graph-Verwaltung, welche verteilt erfolgt, besitzt die Version eine verbesserte Ausführungspipeline gegenüber dem Nexus++. Hierdurch ist es z.B. möglich, eine Applikation mit grobunterteilten unabhängigen Tasks, wie dem *ray tracing*, um den Faktor 194× bzw. 60× mit Hilfe des Nexus# und Nexus++ zu beschleunigen verglichen mit der seriellen Ausführung. Durch eine reine Softwarelösung würde die Parallelisierung dagegen nur einen Geschwindigkeitszuwachs um den Faktor 31× erreichen.

Nexus# übertrumpft Nexus++ hinsichtlich der Skalierbarkeit, was eine feingranulare Unterteilung der Tasks ermöglicht. Hiervon profitieren Applikationen wie die H.264 Videodekodierung, welche feingranulare Task hat, die zusätzlich noch untereinander abhängig sind. Eine rein softwarebasierte Lösung zur Parallelisierung wäre aufgrund des Laufzeit-Overheads langsamer als die serielle Ausführung. Im Gegensatz dazu erzielen Nexus# bzw. Nexus++ jedoch ein Geschwindigkeitszuwachs um den Faktor 7× und 2.2×.

Wie in dieser Arbeit dargestellt, ist der Nexus# aufgrund seiner umfangreichen Rekonfigurierbarkeit ein zeitgemäßer Hardwarebeschleuniger. Er eignet sich für eine breite Palette von Mehrkernsystemen, angefangen bei eingebetteten Systemen bis hin zu komplexen Hochleistungssystemen.

Contents

Acronyms

AP	Application Processor
API	Application Programming Interface
ASIP	Application Specific Instruction-set Processor
CISC	Complex Instruction Set Computer
DAG	Directed Acyclic Graph
DLP	Data-Level Parallelism
DMA	Direct Memory Access
DP	Dependency Pattern
DR	Dependency Resolution
DSP	Digital Signal Processor
DT	Dependence Table
DVFS	Dynamic Voltage and Frequency Scaling
FHD	Full High Definition
FIFO	First In First Out
FLOPS	FLoating-point Operations Per Second
FP	Floating Point
FP16	16-bit (half-precision) Floating Point format
FP32	32-bit (single-precision) Floating Point format
FP64	16-bit (double-precision) Floating Point format
FPGA	Field-Programmable Gate Array
GFLOPS	10^9 FLOPS
GPU	Graphics Processing Unit
HTS	Hardware Task Scheduler
ILP	Instruction-Level Parallelism
IP	Intellectual Property
IPC	Instructions Per Cycle
ISA	Instruction Set Architecture

ITRS	International Technology Roadmap for Semiconductors
LUT	Look-up Table
MB	MacroBlock
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MPSoC	Multi Processor System on Chip
nm	Nanometer
NUMA	Non-Uniform Memory Access
OMPSS	OpenMP Super Scalar
OPENCL	Open Computing Language
OPENMP	Open Multi-Processing
OS	Operating System
PCIE	Peripheral Component Interconnect Express
PL	Programmable Logic
PS	Processing System
RAW	Read after Write
RISC	Reduced Instruction Set Computer
RTS	Run Time System
SADT	Set-Associative Dependence Table
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMP	Symmetric Multi-Processor
SMT	Simultaneous Multi-Threading
SoC	System on Chip
STARSS	Star Super Scalar
TC	Task Controller
TDP	Thermal Design Power
TFLOPS	10^{12} FLOPS
TG	Task Graph
TGM	Task Graph Management
TLP	Task-Level Parallelism
TLS	Thread-Level Speculation
TMU	Task Management Unit
TP	Task Pool

UHD	Ultra High Definition
VHDL	VHSIC Hardware Description Language
WAR	Write after Read
WAW	Write after Write

List of Tables

List of Figures

Listings

Acknowledgment

Firstly, I would like to express my sincere gratitude to my advisor Prof. Dr. Ben Juurlink for the continuous support of my work, for his immense knowledge, patience, and motivation along the past years, which significantly shaped my professional as well as private life.

Further, I would like to thank the jury members of this thesis, Prof. Dr. Jesús Labarta and Prof. Dr. Jeronimo Castrillon. Thank you so much for all the time and effort you put into reviewing this thesis.

I also thank all of my colleagues for their non-technical and technical support, for the stimulating discussions, for the hard work and commitment we did together before deadlines, and for all the fun we have had in the last few years. In particular, I am grateful to Dr. Ahmed Elhossini, Nina Engelhardt, and to Chi Ching Chi who provided expertise that greatly assisted this research.

I also would like to thank Mr. Michael Frank for writing the German translation of the thesis abstract, and Dr. Gervin Thomas for reviewing it. Moreover, I would like to express my sincere gratitude to Prof. Dr. Guido Araujo, Dr. Daniele Bortolotti, and Divino César Luca for taking the time to review the thesis and provide me with valuable feedback. My gratitude goes as well to Dr. José Gracia for his help tuning the Temanejo task debugging tool to my needs.

Last but not the least, I would like to thank my family: my mother Khawla, father Abdel-Aziz, sisters Rania and Lamia, and brothers Ahmed, Mohammad, and Jawad, who trusted me and have always been a source of love and motivation that has never let me down. To my wife Nivin and kids Ajwad and Hashem for bearing with me during this journey and all the joy they brought to my life.

Finally, this research was partially supported by the German Jordanian University (www.gju.edu.jo), the European Community's Seventh Framework Programme

[FP7/2007-2013] under the ENCORE Project (www.encore-project.eu), grant agreement n° 248647, and the LPGPU Project (www.lpgpu.org), grant agreement n° 288653.

Thank you.

Tamer Dallou

1 Introduction

Data processing in computing systems is a multi-step process, starting with fetching inputs from memory, processing them, and (if necessary) writing the outputs back to memory or displaying them on a monitor.

Depending on the domain of interest, a computing system should fulfill certain requirements. In the High Performance Computing (HPC) domain for example, where applications require huge computational capabilities, Central Processing Units (CPUs) and memory systems should be fast enough to meet application demands. For example, to deliver real time results as in the case of high definition video decoding or live video streaming.

In the embedded systems domain, computations must be done within a certain power budget, or should consume as less power as possible. More and more demands are expected from commercial hand-held devices such as smart phones and tablets, where the power budget is limited, as new applications and games are having more content and are becoming more demanding, such as applications for Ultra High Definition (UHD) video capture and display, virtual reality, or augmented reality.

Computer systems have evolved over time. The computational power of some supercomputers in the past has been met or even surpassed by some of today's mobile devices. For example, Nvidia announced early 2015 its "super chip", the Nvidia Tegra X1 [116], and claimed that it is the first mobile chip that has "more horsepower than the fastest supercomputer of 15 years ago; the ASCI Red, which was the world's first teraflops (TFLOPS) system" [116].

The comparison is obviously for marketing purposes, and is not totally precise, since the ASCI Red supercomputer achieved back then 1 double-precision (FP64[1])

[1]Double-precision floating-point format, known asl binary64 as specified by the IEEE 754 standard, where the number occupies 8 bytes (64 bits) in the computer's memory.

TFLOPS, and the Tegra X1 is capable of 1 half-precision (FP16[2]) TFLOPS. Nevertheless, the other facts are still correct and show how amazingly the semiconductor industry has evolved in this relatively short period of time.

Moreover, the ASCI Red supercomputer occupied about 150 square meters and consumed 500 kilowatts plus another 500 kilowatts for cooling the system. The Tegra X1 is about the size of a thumbnail, and consumes less than 10 watts.

Of course, supercomputers have evolved since then, tackling more complex problems in various domains, at high precision. At the time of writing this thesis, the most powerful supercomputer in the Top 500 list [154] is the Sunway TaihuLight at the National Supercomputing Center in Wuxi, China [155]. It has 10,649,600 cores, 1,310,720 GB of memory, and can achieve FP64 93,014.6 TFLOPS, at a massive power budget of 15,371 kilowatts.

The next two sections discuss the developments in the semiconductor industry in the last few decades, and how the trend shifted from designing single core microprocessors, to multicore systems.

1.1 Moore's Law and Dennard Scaling

In 1965, Gordon Moore predicted in his paper [114] that the number of transistors in an integrated circuit would double every 18-24 months. His prediction is based on observing how the electronics manufacturers back then was shaping the semiconductor industry by competing to make their offerings more powerful. After sometime, the prediction itself, also known as Moore's law, started to shape the roadmap of the semiconductor industry [106].

Over the past 50 years, Moore's prediction proved to be true, and the semiconductor industry continued to decrease the transistor size and pack more of them into a single chip. This can be seen in the "Transistors" curve of Figure 1.1, which shows the number of transistors used in some cutting edge microprocessors over time.

[2]Half-precision floating-point format, known as binary16 as specified by the IEEE 754 standard, where the number occupies 2 bytes (16 bits) in the computer's memory.

Since the introduction of the first transistor-based computer back in the 1950s [95] and until the last decade, the main focus for computer architects was improving the single thread performance.

Microprocessor manufacturing technology has been directly reflected on application's performance. On one hand, the smaller size of integrated circuits (IC) components yielded microprocessors running at higher clock frequencies, resulting in faster applications as shown in the "Frequency" and "Single-Thread Performance" curves of Figure 1.1. On the other hand, the increased transistor density enabled the design of improved micro architectures and larger caches.

The amount of dynamic power P dissipated by a microprocessor is given by Equation 1.1 [164], where α is a constant indicating the percentage of the system that is active, C is the equivalent system capacitance, V is the supply voltage, and f is the clock frequency.

$$P = \alpha C V^2 f \tag{1.1}$$

Up the early 2000s, decreasing the transistor size was accompanied by decreasing its supply voltage and increasing the clock frequency, and thus maintaining a fairly constant power density as indicated by Equation 1.1. This effect is known as the Dennard scaling effect [50] (named after Robert Dennard who and his team first described this effect in their paper [50] in 1974). Dennard scaling effect suggests that the scaling of the supply voltage and the clock frequency is proportional to the feature size. In other words, more transistors can be integrated in a smaller area, clocked at a higher speed, and would consume the same or less power.

The clock frequency continued increasing as Dennard et al. suggested from the sub 1-MHz range back in the early 1970s, up to the 3-4 GHz range about 15 years ago and stagnated since then. This stagnation is mainly due to the power dissipation and the resulting heat in integrated circuits. The more power a microprocessor dissipates, the more heat it emits, and more difficult it becomes to cool it down which is a major problem. This problem is generally referred to as the *power wall*, and it is the reason why the microprocessors produced since the early 2000s until today have been designed to dissipate 150 Watts at most. This can be seen from the "Power" curve of Figure 1.1.

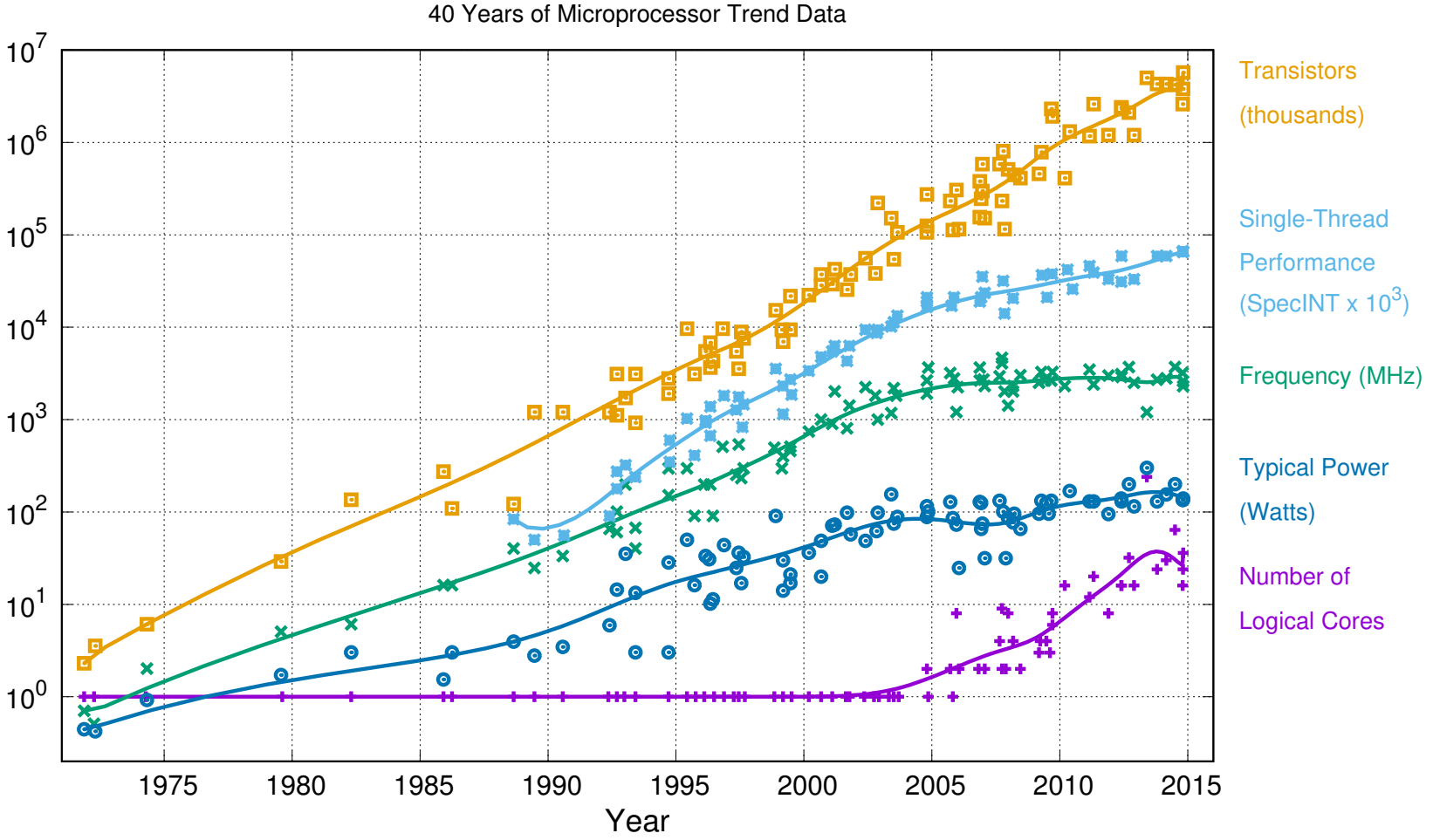


Figure 1.1: The evolution of transistor density, frequency, power, performance, and number of cores in microprocessors over time [C. Moore: Data Processing in Exascale-Class Computer Systems, April 2011]. (Original data up to the year 2010 collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. New data collected for 2010-2015 by K. Rupp [137].)

System capacitance and supply voltage should have scaled down according to Dennard et al, but what happened in reality is far from that. System capacitance increased over time as microprocessor manufacturers used the increased transistor density to add more features, and produced chips of similar of even larger areas [47]. For example, Intel's 386 microprocessor [166] produced in 1985 has 275,000 transistors ($1.5\mu m$ process technology), in a $104mm^2$ die. Almost 30 years later, Intel's i7-5960X microprocessor (Haswell-E micro-architecture) [168] has 2.6 billion transistors ($22nm$ process technology), in a $355mm^2$ die. Server-class microprocessors such as Intel's Xeon Haswell-E5 [168] are even more dense: it has 5.56 billion transistors ($22nm$ process technology), in a $661mm^2$ die.

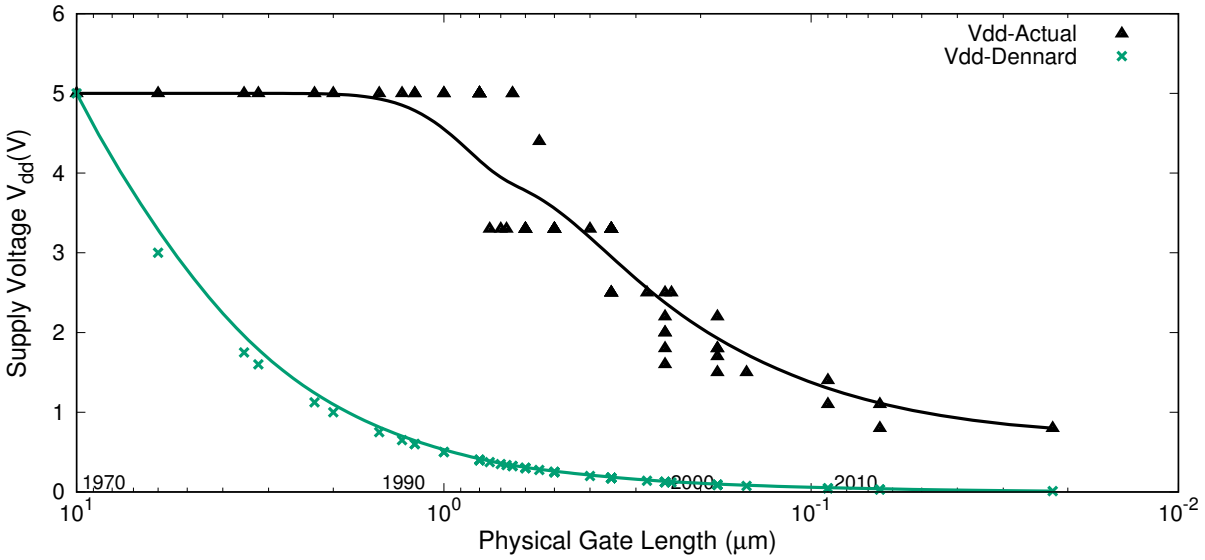


Figure 1.2: Supply voltage scaling vs semiconductor technology [47]. After the early 1990s ($0.65\mu m$ process technology) and until 2005 ($0.035\mu m$ process technology), the supply voltage (V_{dd}-Actual) scaled down at a slower pace that what Dennard et al. suggested (V_{dd}-Dennard). Logarithmic scale of the x-axis to highlight the period after 1990.

The supply voltage used overtime for different transistor technologies is shown in Figure 1.2. Until the early 1990s ($0.65\mu m$ process technology), microprocessors maintained a 5V supply voltage. Back then, power dissipation was not a major issue (Intel's Pentium microprocessor consumed less than 10 Watts [167]). Only between 1990 ($0.65\mu m$ process technology) and 2005 ($0.13\mu m$ process technology), the voltage

started scaling down and at a slower pace than to the process technology as can be seen in Figure 1.2. Calculating the scaling factor of the process technology, and reflecting it on the supply voltage results in the lower curve (V_{dd}-Dennard) of Figure 1.2. This theoretical curve shows the scaling behavior of the supply voltage suggested by Dennard et al. in order to produce more dense microprocessors that consume less power. The upper curve of Figure 1.2 shows how the voltage supply changed in reality, and how far it is from the trend suggested by Dennard et al.

Many techniques to reduce and control power dissipation have been implemented in microprocessors over time, such as dynamic voltage and frequency scaling, underclocking, clock gating, etc. Although these helped in reducing the power density of microprocessors, they did not prevent hitting the power wall eventually. Furthermore, the static and leakage power in semiconductors became more significant as transistor density increased over time [41]. All of the above explain the increase in power consumption of modern microprocessors (made of smaller transistors) compared to the older ones, as shown in the "Power" curve of Figure 1.1.

Nevertheless, transistor process technology kept improving and it still does with more transistors being integrated in smaller chips. Therefore and about 10 years ago, the trend in microprocessor manufacturing changed to use the increased transistor density to integrate more than one core on chip. This trend can be clearly seen in the "Number of Logical Cores" curve of Figure 1.1. The following section sheds some light on the expected future trend in semiconductors based on the *International Technology Roadmap for Semiconductors*.

1.2 The Multicore Era

The years to come are going to be interesting, as the transistor size cannot be reduced anymore due to its physical limits. Moreover, semiconductors manufacturing costs increase as transistors become smaller [141, 151]. This does not mean the end for the semiconductor industry. On the contrary, this has pushed scientists to search for new approaches to further increase the transistor density and consequently the number of cores on chip.

The *International Technology Roadmap for Semiconductors (ITRS)* is a technological roadmap produced periodically by semiconductors experts worldwide. It discusses

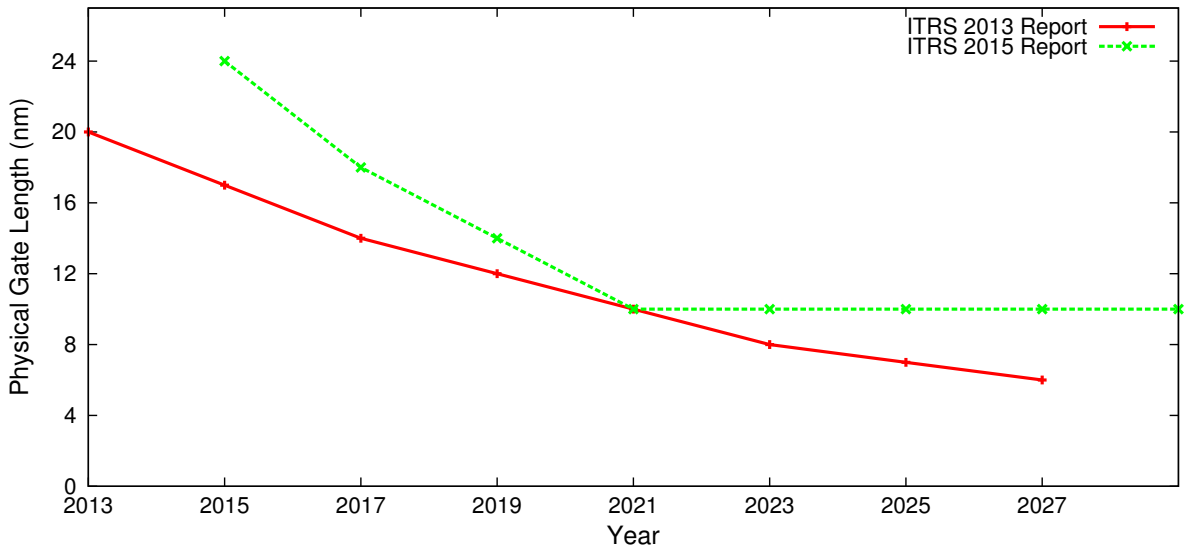


Figure 1.3: Comparing the transistor gate length in semiconductors as predicted by the International Technology Roadmap for Semiconductors in 2013 [140] and 2015 [141].

many aspects related to the semiconductor technology and predicts the trends for 10-15 years in the future. ITRS 2013 and 2015 prediction roadmaps are shown in Figure 1.3.

In 2013, ITRS expected that process technology would have shrunk further reaching $5nm$ in 2027. However, two years later ITRS 2015 roadmap suggested that the semiconductor industry will continue reducing the transistor size down to $10nm$ in 2021 and would stagnate there. Since this implies that the semiconductor industry will be eventually running out of horizontal space in integrated circuits, ITRS suggests that the technology will move instead towards vertical integration, in order to achieve the increased transistor density and integrate more cores on chip.

The ITRS 2015 roadmap also predicts the number of cores to be available in application processors in the coming years as shown in Figure 1.4. ITRS predicts that computers will have more than 36 application processors and 247 graphic processing units in 2025.

Nowadays multicore processors are the standard, not the exception. Table 1.1 lists some representative, off-the-shelf multicore processors that are in production at the

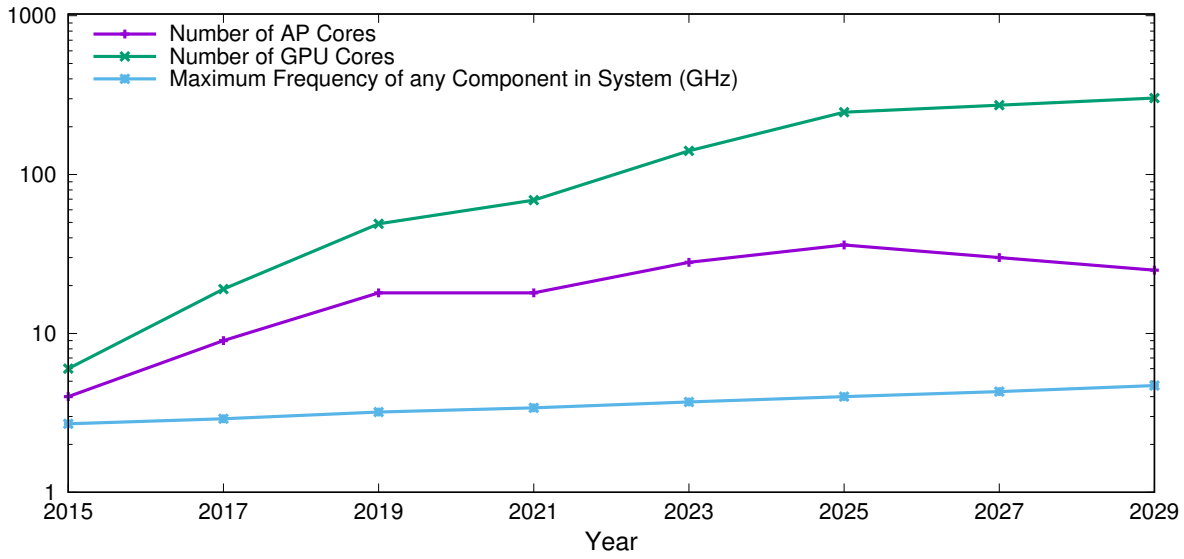


Figure 1.4: Number of application processors (AP) and graphics processing units (GPU) integrated on chip, as predicted by the International Technology Roadmap for Semiconductors (ITRS) in 2015 [141].

time of writing this thesis. The processors listed in the table are the high-end products of major companies and are targeting different domains, ranging from mobile systems such as smartphones and tablets, to desktops and powerful workstations. High end workstation processors have up to 24 cores, such as *Intel's Xeon processor*, while desktop processors have 4 - 10 processors, such as *AMD's Summit Ridge* and *Intel's core i7* processors listed in Table 1.1. Also processors used in mobile systems can have up to 10 cores nowadays, but typically they have 4-8 cores.

According to *statista.com*, more than 1.4 billion smartphones [146] and about 500 million computers (including desktop PCs, laptops, and tablets) [145, 147] were shipped worldwide in 2015. Most of these machines are powered by multicore processors, based on the previously discussed trend shift towards multicores. In order to make use of these systems, the software must be modified to utilize the multiple cores. In other words, if the software is not multicore-aware, the increased number of cores on chip and all the semiconductor technology advancements behind it would be useless.

The next two sections discuss the complexity of programming multicore systems and the major challenges of parallel programming.

Company	Processor Model	# Cores (# Threads)	Process Technology	Clock Speed	Target Market
AMD	Summit Ridge	4 (8)	$14nm$	2.8 GHz	Desktops
Apple	A10 Fusion	4 (4)	$16nm$	up to 2.35 GHz	Mobile
Intel	i7-6700K	4 (8)	$14nm$	4.0 GHz	Desktops
Intel	i7-6950X	10 (20)	$14nm$	3.0 GHz	Desktops
Intel	i7-7500U	2 (4)	$14nm$	2.7 GHz	Notebooks
Intel	Xeon E7-8890V4	24 (48)	$14nm$	2.2 GHz	Workstations
Mediatek	Helio X20	10 (10)	$20nm$	1.4 - 2.34 GHz	Mobile
Nvidia	Tegra X1	8 (8)	$20nm$	1.3 - 1.9 GHz	Tablets
Qualcomm	Snapdragon 821	4 (4)	$14nm$	1.6 - 2.35 GHz	Mobile
Samsung	Exynos 8890	8 (8)	$14nm$	1.6 - 2.3 GHz	Mobile

Table 1.1: Representative off-the-shelf multicore processors from major microprocessor vendors in 2015/2016, listed alphabetically.

1.3 Application Parallelism

The goal of parallel programming is to break down an algorithm/application into smaller parts that can fully exploit the multicore architecture, while maintaining correctness and efficiency [17]. Transforming a sequential algorithm to a parallel one, according to Solihin [144], includes the following steps:

- decomposition: decomposing the problem to smaller tasks, the smallest exploitable units of concurrence,
- assignment: tasks are assigned to processes,
- orchestration/coordination: handling all work related to data movement, communication and synchronization between processes,
- mapping: scheduling processes on the available processors.

Before discussing the decomposition step, I will discuss briefly Flynn's classification of computer architectures [59], which is related to the assignment, orchestration/coordination and mapping steps in the above list.

Michael Flynn proposed in 1966 four classifications of computer architectures based on the number of data streams and instruction streams in the architecture. Flynn's taxonomy of computer architectures is summarized in Table 1.2.

	Single-Instruction	Multiple-Instruction
Single-Data	**SISD** (Sequential Computers)	**MISD** (Redundancy, Fault Tolerance)
Multiple-Data	**SIMD** (Vector Processors, GPUs)	**MIMD** (Modern Multicores)

Table 1.2: Flynn's classification of computer architectures.

In the single-instruction-multiple-data stream (SIMD) architecture, multiple processing elements perform the same operation on multiple data streams in parallel. This is mostly common in vector processors and graphics processing units (GPU) where operations are naturally parallel, for example adjusting the brightness of a picture, in which the desired brightness value should be written for each pixel of the picture.

The multiple-instruction-multiple-data stream (MIMD) architecture, where multiple processing elements can perform different operations on different data streams concurrently, is the most interesting one for the scope of this thesis. This class of parallel architectures is where most modern multicore systems fit, and the most difficult to program.

The remaining two types are less relevant for the scope of this thesis: single-instruction-single-data stream (SISD) architecture which is basically a sequential processor, and multiple-instruction-single-data stream (MISD) processors which can be found in fault tolerance systems, where multiple processors are running the same program and the final result would be chosen based on a voting algorithm for example.

MIMD architectures can be further categorized based on their memory system. It can be a distributed memory, or a shared memory system among the different processors. Shared memory can either be uniformly or non-uniformly accessed by the processors. The non-uniform memory access (NUMA) architecture has become more common in multicore systems [67, 85, 89] and adds more complexity to parallel programming, since not only access to shared resources must be coordinated, but also mapping application parts to the available processing cores plays a crucial role in performance, as it affects data locality and cache hit rates.

To obtain speedups from multicore architectures, applications need to use parallel algorithms that can efficiently exploit the underlying multicore architecture. This is not a trivial task, since the application should be broken down to smaller parts and parallelism should be detected, application parts should be efficiently mapped to the available resources, dependences must be maintained, synchronization must be handled, and finally after having all the different parts finished, their results should be collected in order to produce the final application result. Therefore, due to these management aspects, parallel algorithms are more challenging to develop than their sequential alternatives.

Going back to the list of steps for transforming a sequential algorithm to a parallel one shown at the beginning of this section, the decomposition step states that the algorithm/application should be broken down to the smallest exploitable unit of concurrence, in order to achieve the maximum speedup. An application can have one or a mix of the following types of parallelism,

- Data parallelism: which is related to SIMD architectures, where a single operation is performed concurrently on multiple data points. This is usually the easiest type of parallelism and typically yields high speedups, as the degree of concurrency depends on the input data size. For example, a program that sums two arrays of size N and stores the result in a third array is a straightforward example of data parallelism since all addition operations can be performed in parallel. In this case, a multicore system of N cores (or more) can achieve $N\times$ speedup with respect to serial execution.
- Pipeline parallelism: typical of loops, where loop iterations are partially dependent (there exists a loop-carried dependence). In this case, the goal is to start iteration k as soon as the loop-carried dependence data is ready from a previous iteration, for example *k-1*, without waiting for the whole loop *k-1* to finish. This is done by dividing the operations in the loop into stages so that independent stages can run in parallel.
- Task parallelism: this type is related to MIMD architectures, where different programs are performed on different or same sets of data. For example, when performing statistical analysis on a certain set of data, such as calculating the average, geometric mean, min and max, etc. If there are dependences between the operations, this type of parallelism can be difficult to detect and exploit.

The speedup gain is less than in data parallelism, as it depends on the number of parallel tasks in the application.

In modern programming models, the programmer can explicitly define any code block (a function, loop, or arbitrary part of the code) as a task which can run in parallel to other tasks. Task parallelism is therefore key to utilize multicore systems and given that many algorithms can be parallelized this way, task parallelism is the main type of interest for this thesis work.

1.4 Parallel Programming Challenges

The multicore era brought four major challenges for the parallel programming community. The first challenge is the *detection of parallel regions*. Classically, detection of parallel regions is handled manually by the programmer or automatically by the compiler [86, 94, 120]. Despite the existence of libraries [45, 101] and programming language extensions [62, 73, 134, 157] to help the programmer parallelize programs, parallel programming is considered an error-prone task [10, 76, 126].

While manual parallelization is limited to simple algorithms and a small number of cores, compilers are able to tackle more complex scenarios [13, 30, 44, 119, 126, 131]. Nevertheless, there are some cases where compilers cannot automatically parallelize the code, for instance when the program or target region makes heavy use of pointers and a precise alias analysis or dependence analysis is not possible [10, 86]. A major example is parallelism in loops, where two common cases exist [10, 86, 90]: (1) due to dependences in loops that cannot be resolved at compile time compilers cannot detect the existence of loop-carried dependences, and thus they give up on parallelizing the loop. Since at runtime such dependences might never happen (referred to as *dynamic-DOALL loop*); using task parallelism to divide loop iterations into tasks can effectively handle parallelizing such loops [57]. (2) Even if the compiler can detect the loop-carried dependence(s) at compile-time (so the loop is called a *DOACROSS* loop), there is no effective loop parallelization algorithm for *DOACROSS* loops that is always effective [76, 90, 103, 126]. Task parallelism can effectively handle this case as well.

The second challenge in parallel programming is the *management of tasks and resources* including: *scheduling/mapping* of ready tasks to the worker cores and *coordi-*

nation of access to the shared resources: how to efficiently handle this challenge? In the case of MIMD architectures and task parallelism as discussed before, decomposing an algorithm manually (by the programmer) or automatically (by the compiler) into smaller tasks and orchestrating access to the parallel architecture can be complex.

The third challenge is *performance portability*: how to make the applications maintain high performance over a wide range of parallel architectures? Multicore processors are rapidly evolving and are different from one another, although most of modern processors fit under either SIMD or MIMD architectures according to Flynn's taxonomy. Nevertheless, parallel programmers / programming tools should take performance portability into account in order to avoid code rewriting for each architecture.

The fourth and last challenge is *scalability*: can the application benefit from the increased number of cores? Amdahl's law [5] shown in Equation 1.2, named after computer scientist Gene Amdahl, gives the theoretical speedup S in latency to be expected when executing part of the application in parallel (i.e. only part of the application can benefit from the increased number of cores). Amdahl's law is given as a function of the number of the available cores N and the percentage P of the application that can be executed in parallel.

$$S = \frac{1}{(1 - P) + \frac{P}{N}} \tag{1.2}$$

For example, if 90% of an application can be run in parallel, the expected speedup when running this application on a 16-core machine equals $1/((1-0.90)+(0.90/16)) = 6.4\times$. Furthermore, this application's speedup cannot be more than $10\times$ regardless of the number of cores available.

In the ideal case, an application that can be entirely divided into many smaller tasks has a linear scalability. This means that it can be sped up by $2\times$ when running on 2 cores, $4\times$ when running on 4 cores and so on, up to the maximum number of tasks it can be divided into. Applications vary in their scalability, from those that scale well (near-linear scalability), also known as embarrassingly parallel applications, to the inherently serial applications that do not scale at all and can use only one core.

Figure 1.5 depicts the expected speedup according to Amdahl's law, varying the percentage of the application that can run in parallel and the number of the cores. Amdahl emphasizes that the serial part of an application limits its scalability and the

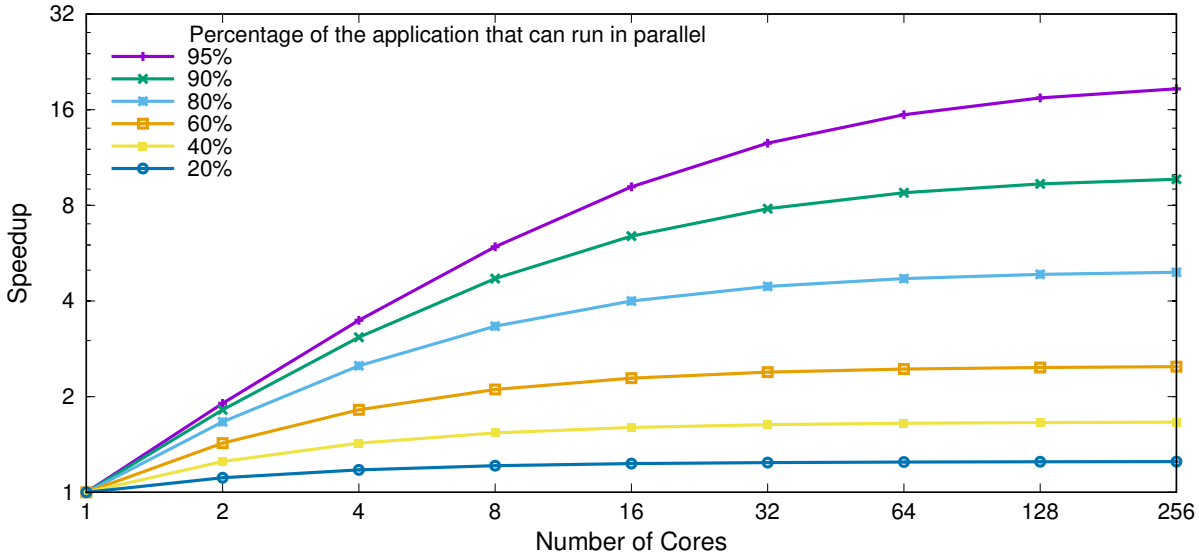


Figure 1.5: The theoretical speedup in latency to be expected when executing part of the application in parallel, according to Amdahl's law. It can be seen that the serial part of the application is limiting its scalability. The y-axis in logarithmic scale.

desired speedup. This can be clearly seen in Figure 1.5, especially for applications that have more than a 10% serial part. As the number of cores on chip is continuously increasing, application developers and parallelization tools should pay extra attention to Amdahl's law and exploit fine-grain parallelism in order to achieve better scalability.

Given the increasing complexity and diversity of multicore systems, the increasing complexity and demand of applications and the importance and susceptibility of applications scalability, manual parallel programming is not feasible, as productivity is limited and performance portability can be compromised. Therefore, it is important to have a high level parallel programming model that well-exploits the available resources, enables portability among a wide range of multicore systems, provides a high degree of scalability, and also helps improving programmers productivity.

Indeed *productivity* is another challenge for parallel programming, depending on whether the programmer has to manually extract parallelism in the application, explicitly exploit the hardware resources and handle coordination and synchronization between application parts.

The next section presents some parallel programming models that aim at solving these challenges, with focus on task-based programming.

1.5 Parallel Programming Models

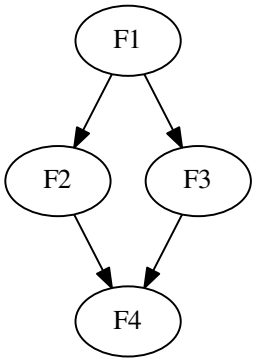
Several programming models have been proposed to enable parallel programming, but they differ in their target architecture, application domain, and usability. Examples include Cilk [11, 22, 62], Cilk++ [43, 96], Google's MapReduce [49], POSIX threads [113], Intel's TBB [136], OpenCL [87], CUDA [115], C++AMP [68], OpenMP [23, 45], StarPU [12], StarSs [127] and OmpSs [54].

Although they differ in the degree of abstraction they provide, all parallel programming models share the goal of decoupling the programmer from the underlying multicore system. Low level abstractions such as POSIX threads, CUDA, and OpenCL require the programmer to have a thorough knowledge of the underlying architecture and to explicitly manage parallelism and handle coordination between the available resources. Therefore, low level abstractions are difficult to use and therefore have limited productivity. Moreover, if the target application has a complex dependence pattern, the scalability of such abstractions is totally dependent on the programmer's ability to extract and manage parallelism, which is a difficult task for the average programmer. Moreover, CUDA, OpenCL, C++AMP, and OpenACC target the application domain where massive data parallelism exist, and therefore are more useful to use in SIMD/GPU architectures.

Higher-level abstractions such as Intel's TBB, C++AMP, Charm, Cilk/Cilk++, OpenMP, StarPU and StarSs/OmpSs are easier to use and typically lead to easier programmability and thus higher productivity. They sometimes also imply higher overheads as their runtime systems need to do more work to bridge the gap between the programming model concepts and the machine's specific architecture. High level abstraction are also different from one another when it comes to programmability and productivity. Some of them (Intel's TBB, C++AMP, Charm, Cilk/Cilk++) require the programmer to change the application code, use new syntax and take care of synchronization between the parallel application parts. Other easier-to-use abstractions such as OpenMP, StarPU and StarSs/OmpSs require the programmer to add some

```
1  // Functions' definitions
2  void main(){
3   int A, B, C, D, E;
4
5   #pragma omp task
6   F1(A, B);
7
8   #pragma omp taskwait
9
10  #pragma omp task
11  F2(A, C);
12
13  #pragma omp task
14  F3(B, D);
15
16  #pragma omp taskwait
17
18  #pragma omp task
19  F4(C, D, E);
20 }
```

(a)



(b)

Figure 1.6: Simple OpenMP example showing (a) source code with task annotations and (b) the task graph.

hints to the code in the form of pragmas, which later on can be converted to runtime system calls so as to exploit the parallelism in the application.

A simple source code of a task-based program in OpenMP is shown in Figure 1.6(a), along its task graph in Figure 1.6(b)

In the listing shown in Figure 1.6(a), the programmer defines four tasks (F1 to F4) using the *#pragma omp task* directive. While F2 and F3 can run out of order and in parallel, they must wait for F1 to finish due to the *#pragma omp taskwait* directive. Similarly, task F4 must wait for tasks F2 and F3 to finish execution before it can start. This results in the task graph shown in Figure 1.6(b).

The task directive can be placed anywhere in the program annotating a function, a part of a loop body, or any arbitrary code block. Whenever a task directive is encountered by the thread executing the main program, a task is spawned. Upon

spawning a task, it will placed in a conceptual task pool and its execution will be deferred until a later time. The worker threads will execute the tasks in the task pool until the pool becomes empty.

There are more directives in OpenMP and other task-based programming models providing the programmer with the means of covering different parallel scenarios. However, the programmer is responsible for discovering and managing parallelism.

In my opinion, the winner programming model is the one that:

- handles exploitation and management of the underlying system, relieving the programmer from explicitly dealing with the parallel architecture details, and thus increasing productivity,
- detects and manages parallelism in the application, and decomposes it into smaller parts accordingly, with minimum or no intervention from the programmer, also increasing productivity,
- provides performance portability to a wide range of architectures,
- achieves maximum application scalability,
- targets several types of parallelism: data, task, and pipeline parallelism.

The following section discusses the dependency-aware task-based programming model that targets increased programmer productivity through minimizing the programmer's role in parallelism extraction and management, and thus is a key candidate that tackles the main challenges in programming multicore systems.

1.6 The Dependency-Aware Task-Based Programming Model

The dependency-aware task-based programming model requires minimum intervention from the programmer, by annotating sections of code that can potentially run in parallel (tasks) with the conditions under which execution is allowed (dependences). This programming model is very promising, as it provides means for solving the main parallel programming challenges discussed previously. The runtime system then organizes execution of all tasks respecting the constraints, avoiding the need for the

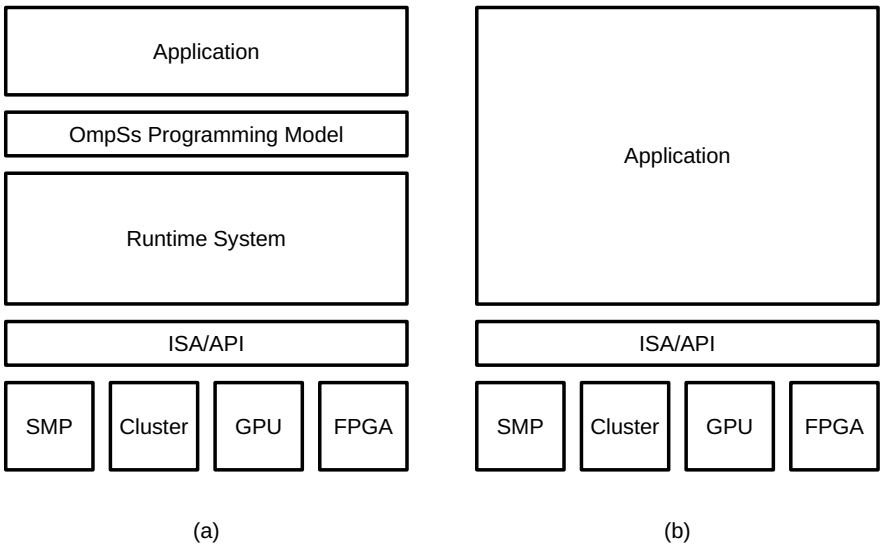


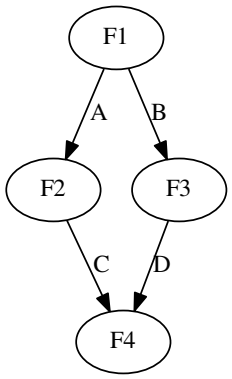
Figure 1.7: (a) OmpSs high-level abstraction of hardware from the application's point of view, enabling the programmer to focus on the application logic and leaving the hardware resources specifics and parallelism management to the runtime system, which in contrast to (b), those should be included in the application in addition to the application logic [31].

programmer to reason (and potentially make difficult-to-find mistakes) about the circumstances when the dependences are fulfilled.

OpenMP v4.x [23], StarSs [127] and OmpSs [54] are good examples of this model. The runtime system of those programming models builds, at runtime, a task graph based on the sequence of function calls and their input/output requirements and determines which tasks are ready to run. This model gained a special importance, since it also targets irregular parallelism, based on the data dependences between tasks, while at the same time, it requires minimal intervention from the programmer.

OmpSs high-level hardware-software layers can be seen in Figure 1.7(a), decoupling the application from the hardware, with the runtime system orchestrating the usage of the hardware resources transparently to the programmer. In contrast to OmpSs, Figure 1.7(b) shows the classical, low-level abstraction of the hardware, where the programmer should include hardware resources specifics and explicitly manage parallelism in addition to the application logic in order to make use of the different hardware resources.

```
1  // Functions' definitions
2  void main(){
3   int A, B, C, D, E;
4
5   #pragma omp task inout(A) out(B)
6   F1(A, B);
7
8   #pragma omp task in(A) out(C)
9   F2(A, C);
10
11  #pragma omp task in(B) out(D)
12  F3(B, D);
13
14  #pragma omp task in(C) in(D) out(E)
15  F4(C, D, E);
16 }
```



(a) (b)

Figure 1.8: Simple OmpSs example showing (a) source code with inline task annotations and (b) the task-dependency graph.

OmpSs enables the programmer to run his/her code on a heterogeneous machine, by specifying the target of each task: a symmetric multiprocessor (SMP) system, cluster, GPU, or a FPGA. Chapter 2 includes more information about the OmpSs programming model.

The same simple program shown previously in Figure 1.6 is re-written using OmpSs dependency-aware task annotations as shown in Figure 1.8. There are four function calls in the main program to the F1, F2, F3, F4 subroutines, same as in the OpenMP example before. Knowing that F1 reads the variable A and writes its outputs to the variables A and B, the programmer can annotate subroutine F1 by the OmpSs task pragma inline just before calling F1 in the program as indicated in line number 5 in the listing shown in Figure 1.8(a). Annotations of the F2, F3, and F4 subroutines are done similarly in lines 8, 11, and 14 respectively.

Whenever an annotated subroutine is called, rather than being directly executed, a task will be created by the OmpSs runtime system and added to the application task graph based on its input/output parameters. In Figure 1.8(b), the task graph

of the simple example is depicted. The dependences between tasks are interpreted by running F1 first, and after it finishes execution, the runtime system launches F2 and F3 out of order and in parallel. Once F2 and F3 finish their execution, the runtime system launches F4.

By launching tasks according to the task graph, OmpSs ensures that the program runs correctly and produces the correct results, same as those resulting from the serial execution of the program. In contrast to the OpenMP[3] example shown in Figure 1.6, using the dependency-aware task directive in OmpSs the programmer does not need to think about dependences between the different tasks, resulting in an increased productivity.

```
1 void Cholesky(int NT, float *A[NT][NT])
2 {
3    int i, j, k;
4    for (k=0; k<NT; k++)
5    {
6       #pragma omp task inout (A[k][k])
7       potrf(A[k][k]);
8       for(i=k+1; i<NT; i++)
9       {
10         #pragma omp task in (A[k][k]) inout (A[k][i])
11         trsm(A[k][k], A[k][i]);
12      }
13      for(i=k+1; i<NT; i++)
14      {
15         for(j=k+1; j<i; j++)
16         {
17            #pragma omp task in (A[k][i], A[k][j]) inout (A[j][i])
18            gemm( A[k][i], A[k][j], A[j][i]);
19         }
20         #pragma omp task in (A[k][i]) inout (A[i][i])
21         syrk(A[k][i], A[i][i]);
22      }
23   }
24 }
```

Listing 1.1: OmpSs example of Cholesky.

Another example of application parallelism using OmpSs is shown in Listing 1.1, which is the source code of the Cholesky algorithm [39, 123], a matrix decomposition

[3]OpenMP supports task dependencies starting from v4.0 [23].

algorithm (for real symmetric and (Hermitian) positive-definite matrices) widely used for solving systems of linear equations. Except for lines 6, 10, 17, and 20, the code is sequential. It has nested loops and calls to four subroutines *(potrf*[4]*, trsm*[5]*, gemm*[6] *and syrk*[7]*)*. Code lines 6, 10, 17, and 20 in Listing 1.1 are what the programmer needs to add to the sequential code in order to parallelize it: the *#pragma* directives, indicating the programmer's hints about the subroutines he/she wants to run as tasks.

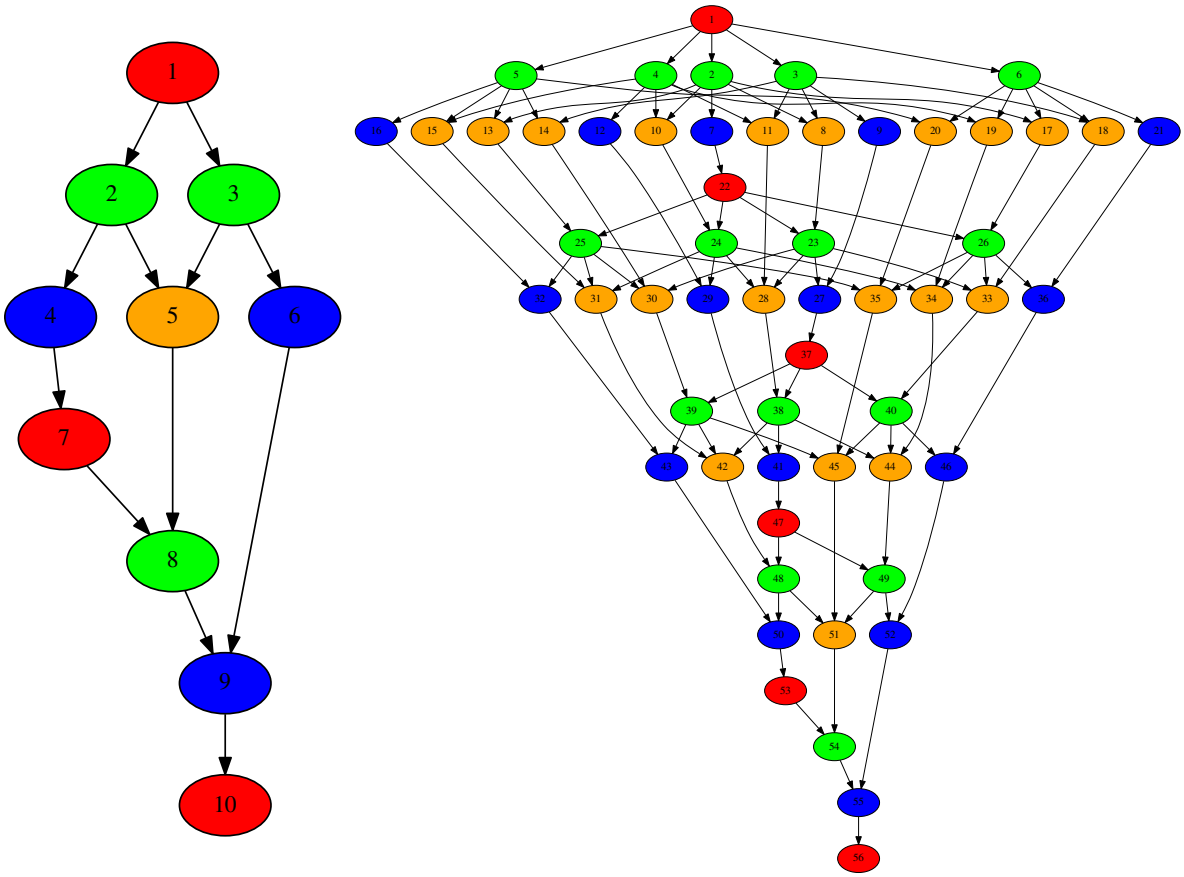
Based on these directives, the runtime system dynamically generates the dependency graph (also know as the task graph) of all tasks. Two task graphs for Cholesky are shown in Figure 1.9, (a) for a small matrix size (3x3), and (b) for a bigger matrix (6x6). The nodes in the graphs are colored to reflect the different subroutines in Listing 1.1. Based on the task graph, the runtime system can run the tasks out of order, as long as dependences between tasks are preserved. This type of parallelism enables extraction of irregular parallelism in the application. Task graph (a) in Figure 1.9 shows that the application starts with task 1 running *potrf*, then tasks 2 and 3 running in parallel *trsm*, then tasks 4, 5, 6 running in parallel instances of *syrk* and *gemm*, finally tasks 7 to 10 running sequentially because of data dependences between them. Task graph (b) in Figure 1.9 shows that for a different input size, the resulting task graph has more parallelism, but also becomes more complex, and thus more challenging for the runtime system to maintain.

One more thing worth mentioning about the difference between task graphs (a) and (b) of Figure 1.9, is the number of tasks depending on a certain task, for example (the green nodes) in the second level, which are dependent on the first task (the red node) in the first level. This number increases with input size and can grow large in the case of Cholesky, and the runtime system has to deal with such a challenge.

The Cholesky's OmpSs source code shown in Listing 1.1 and the task graphs shown in Figure 1.9 reveal the challenges the runtime system has to deal with:

- the subroutines in program can have an arbitrary number of arguments,
- tasks can have an arbitrary number of dependent tasks,

[4]Computes the Cholesky factorization of a symmetric positive-definite matrix.
[5]Solves a triangular matrix equation.
[6]Computes a matrix-matrix product with general matrices.
[7]Performs a symmetric rank-k update.



(a) Input size 3x3

(b) Input size 6x6

Figure 1.9: Task graphs (generated by *TEMANEJO* [24]) of the OmpSs-parallelized Cholesky algorithm [39, 123] decomposing a matrix of size (a) 3x3, and (b) 6x6. Nodes correspond to task instances, their colors refer to the respective functions in Listing 1.1, and edges indicate dependences between tasks. Tasks are numbered according to the invocation order, but those in the same level can run concurrently, out-of-order.

- tasks can have an array as an input/output/inout, which complicates dependence discovery in the case of partial overlapping of memory regions used by different tasks,
- parallelism can be distant. For example in Figure 1.9b, task 21 is ready to run after task 6, but 15 tasks are submitted in between. This means that the runtime system must insert those tasks to the task graph before figuring out that task 21 is ready to run after task 6. This is important in cases when the runtime system is blocked by for example a memory or task graph limit, before inserting the distant task (task 21 in the example).
- task graphs can grow very big and complex.

Moreover, another challenge that is not shown in the Cholesky example is nested tasks where hierarchical task graphs should be maintained.

1.7 Runtime System Overhead

Although dependency-aware task-based programming models such as OmpSs enable parallelism with minimal programmer intervention, the overhead of the runtime system, however, is a decisive factor whether to use such a programming model or not [7, 16, 20, 57, 91, 109, 124, 143, 160]. In the case of OmpSs, the runtime overhead is proportional to the number of tasks an application is broken down to; more tasks implies more coordination and management work that has to be done by the runtime system, and hence an increased runtime overhead. Therefore, programmers tend to group several tasks together to reduce the total number of them, and the runtime system overhead accordingly [7].

The runtime system in the worst case works as a sequential part of the application execution, and can then introduce a scalability bottleneck which follows Amdahl's law. In fact, assuming that the overhead R is a percentage of the application's serial time T, then the parallel application time equals $T/N+RT$, where N is the number of available cores. The expected speedup S can be then calculated as shown in Equation 1.3.

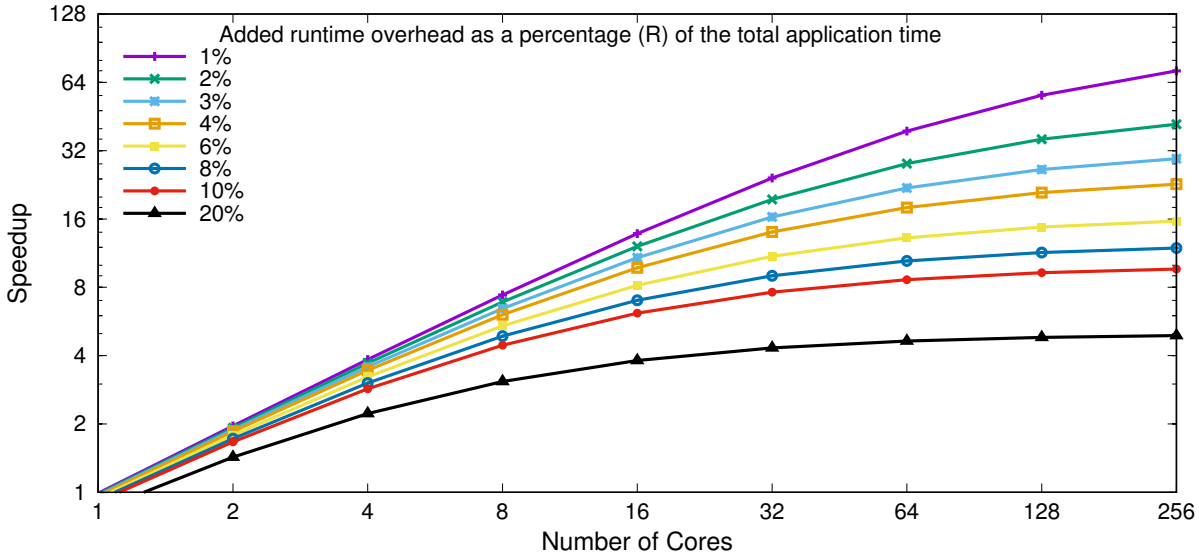


Figure 1.10: The theoretical speedup in latency to be expected when the runtime system overhead, shown as a percentage of the application serial execution time, is serialized with the application execution. The y-axis is in logarithmic scale.

$$S = \frac{T}{\frac{T}{N} + RT} = \frac{1}{\frac{1}{N} + R} \tag{1.3}$$

The expected speedup when varying the number of cores and the runtime overhead time relative to application's time is shown in Figure 1.10. For example, if the overhead is 2% of the application time, then this application can achieve only a 19.5× speedup on 32 cores. Equation 1.3 and Figure 1.10 assume a worst-case-scenario that the overhead is serialized to the application execution, i.e. none of the cores is performing any part of the application while management work is taking place. By management work, as described before I'm referring to the work not related to the application kernel, but to the work needed to run the application in parallel on a multicore processor. This includes spawning tasks, handling dependences and synchronization between them, running them on the available resources. Of course, runtime systems developers are aware of this scalability bottleneck, and they try to hide it by performing the

management work in parallel to the application execution. Nevertheless, Figure 1.10 shows several aspects when the worst-case-scenario might be reached, namely:

- If the resulting tasks are fine-grain, meaning that the worker cores are going to finish their assigned work earlier and ask the runtime system for new work. Consequently, it is more likely that the worker cores will be idle, waiting for work, and hence reaching the worst-case scenario at some points of time during application execution. How often this can happen depends on how fine-grain the tasks are, and how heavy the runtime library is. Moreover, a larger number of tasks implies more runtime overhead, and therefore an increased R in Figure 1.10.
- If the parallelism in the application is limited or its dependency pattern is complex, this increases the probability of starving worker cores as described in the previous point. The more cores there are, the more critical the runtime overhead becomes as shown in Figure 1.10.

The next section shows an example on the effect of runtime overhead on application scalability, and consequently motivates for this thesis work.

1.8 Motivation

In the previous sections, I discussed the evolution of processor architectures and the trend shift towards multicore architectures. Parallel programming challenges are also discussed, and dependency-aware task-based programming such as OmpSs are highlighted as a promising approach that eases programmability of multicore systems, and thus increases programmer's productivity.

Although the idea behind OmpSs, and dependency-aware task-based programming in general, is very promising as a solution for the programmability problem in the multicore era, the overhead of the runtime cannot be neglected, potentially being a bottleneck that limits the scalability of such systems. The overhead can be hidden if the runtime system succeeds to keep the cores busy and does its task graph management responsibilities concurrently. This becomes more difficult with larger numbers of worker cores and/or with applications having fine-grain tasks, complex dependency patterns, or both.

This has been shown in previous works [109, 123] in the case of applications parallelized using StarSs. There, it has been shown that in order to hide the runtime system overhead, the tasks should be of a relatively big size.

In the case of Cholesky shown in Listing 1.1, tasks are coarsened by assigning a block of data points to a single task rather than a single data point. This method is common in linear algebra algorithms such as matrix multiplications and Cholesky factorization [29, 123] and is known as blocking or tiling of data. This leads naturally to grouping of the operations related to all data points in the block, and thus coarsening of tasks. Coarsening of tasks is not always easy (does not require algorithmic change by the programmer) as it is the case in Cholesky and other linear algebra algorithms.

H.264 video decoding [165] is an example of complex applications, where the programmer must modify the source code in order to produce coarse-grain tasks [7, 110]. Snippets of the source code and the task graph are included in the next chapter that discusses OmpSs programming model in more detail, but here I'm discussing the scalability of H.264 video decoding as a representative, driving application for this thesis work. H.264 video decoding is a compelling application, since it is widely used, it has a complex dependency pattern and distant parallelism, and it has fine grain tasks.

Figure 1.11 shows the speedup achieved when running the H.264 video decoding application [7] on a 40-core Xeon E7-4870 multicore machine, running at 2.40GHz. The different curves in the figure show the different configurations used in the experiment. *1x1* means that only 1 macroblock is mapped to each task. In this configuration, the programmer does nothing more than just annotating the sequential code, and this is the target of dataflow programming models in general. As shown in the figure, this configuration does not scale at all. On the contrary, it runs significantly slower than the sequential version.

This is mainly due to the runtime overhead of managing the task graph on one hand, and to the fact that this application has a complex dependency pattern and thus limited amount of parallelism on the other hand. For this reason, and in order to use the multicore system and at the same time achieve some speedup, the programmer needs to group several tasks together to form larger tasks, and decrease the total number of tasks, thus decreasing the overhead of the runtime. This process is done in the case of H.264 video decoding by grouping several macroblocks together and assigning them to a single task to decode them.

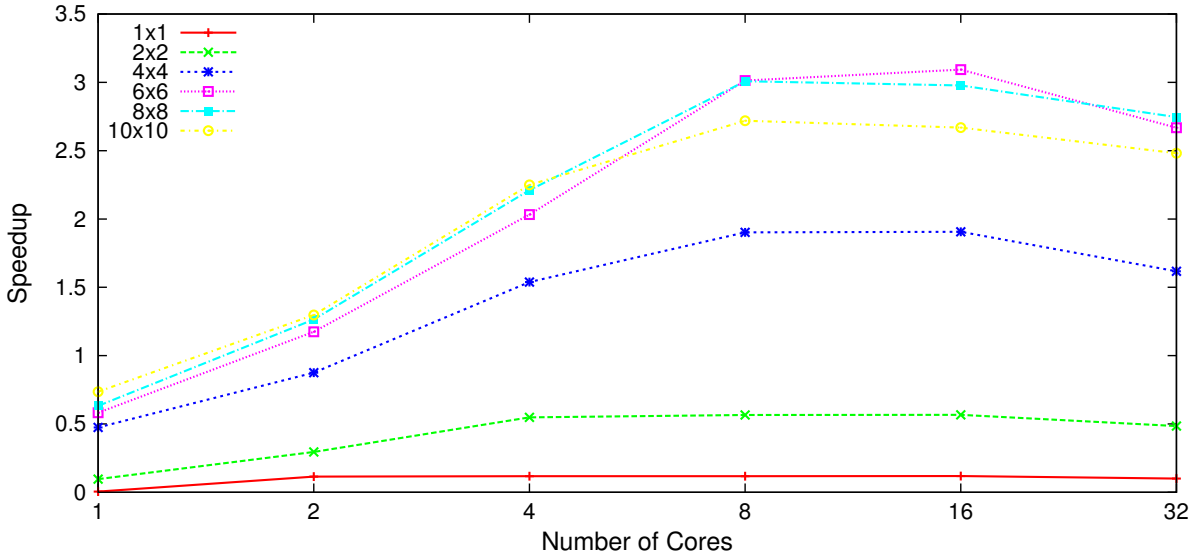


Figure 1.11: The speedup achieved when running OmpSs-H264dec to decode a FHD video sample. Speedup measured against the serial execution time on the same a Xeon E7-4870 machine.

In Figure 1.11, *2x2* means that 4 macroblocks are grouped to be processed by one task. *4x4* means 16 macroblocks are grouped per task and so on. Table 1.3 shows the effect of grouping macroblocks on the total number of tasks and the average task size. The average task size in the simplest configuration is less than $5\mu s$. In the most coarsened configuration (*8x8* macroblocks per task), the average task size is about $190\mu s$. The huge difference of the total number of tasks is also clear.

The grouping process in the case of H.264 is thoroughly presented by Andersch et al. [7]. The authors emphasized that this process is not trivial, as it requires the programmer to explicitly define which macroblocks can be grouped together depending on

	# tasks	total work (ms)	avg task size (μs)
h264dec-1x1-10f	139961	640	4.6
h264dec-2x2-10f	35921	550	15.3
h264dec-4x4-10f	9333	519	55.6
h264dec-8x8-10f	2686	510	189.9

Table 1.3: H.264 video decoding application durations of the different configurations. Durations are obtained from traces collected on Xeon E7-4870. Test input is the first 10 FHD frames of pedestrian_area.h264.

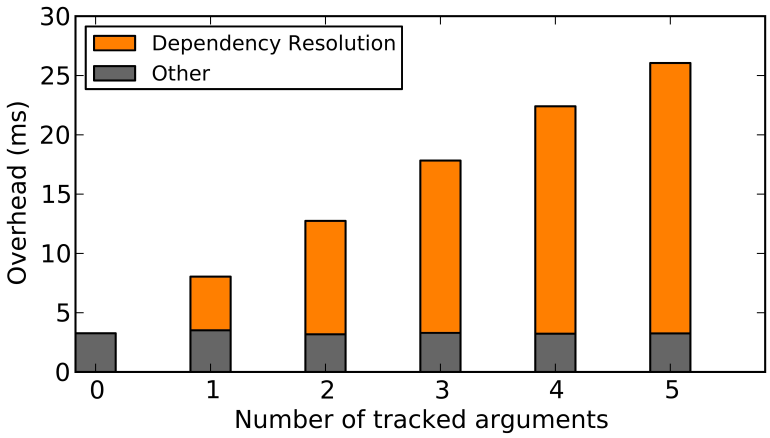


Figure 1.12: Dependency resolution and other runtime overhead for tasks with different number of parameters per task.

the dependences between them. This brings us back to the programmability problem, which limits the usability of OmpSs and similar programming models.

In order to analyze the runtime overhead, several micro benchmarks have been designed that share the same number of tasks, but differ in the number of parameters per task.

The runtime overhead includes:

1. the time spent by the runtime at the beginning of a task's life cycle to:
   a) allocate the task's descriptor,
   b) insert the task to the task graph and decide on task's readiness,
   c) schedule the task to run if found ready.
2. In addition to the time spent by the runtime system at the end of a task's life cycle to:
   a) update the task graph upon its execution completion,
   b) delete the task's descriptor.

The sizes of parts 1.b and 2.a of the runtime overhead in the list above are proportional to the number of arguments per task, as the arguments constitute the basis upon which the task graph is built and maintained.

In one experiment, all tasks have been designed to be independent, in which case part number 2.a of the runtime overhead in the above list in minimized. Varying the number of parameters per task from 0 to 5, the runtime overhead for running those micro benchmarks is shown in Figure 1.12, which shows the aggregate overhead measured on a Xeon E7-4870 multicore machine. It is clear that the overhead is mainly due to maintaining the task graph and tracking the dependences between tasks (overhead parts 1.b and 2.a) for the case of the micro benchmarks with 1 or more arguments per task. The proportion dedicated to parts 1.b and 2.a is highlighted and is directly proportional to the number of parameters a certain task has as shown in the figure. Figure 1.12 also shows that the other parts of the runtime overhead related to task descriptors allocation and deletion and to task scheduling are less impacted by the number of parameters per task.

Therefore, the main goal of this thesis is to improve the scalability of multicore systems, by utilizing fine-grain tasks. This can be done by reducing the runtime overhead of dataflow task-based programming models, by introducing hardware acceleration to the runtime system, which is the research problem addressed in this thesis.

This will result in the following benefits:

- better scalability of applications,
- simple parallel implementation of complex applications: programmers do not need to think about the granularity of tasks and grouping of them and at the same time managing dependences and ensuring correct functionality.

Moreover, using a dedicated hardware unit for task management leads to an improved power efficiency, since multiple cores can run a certain application at a lower frequency and yet deliver the desired results, in comparison to stressing one or two cores at high frequency. Power consumption has not been quantified in this thesis though.

VSs [55] is another research project of the Embedded Systems Architecture group at TU-Berlin, which presents a runtime library for the OmpSs programming model. Compared to the native runtime system of OmpSs, it lacks the instrumentation features

and supports only symmetric multicore architectures on one hand, but this makes it a lightweight runtime system on the other hand. It has been developed in close relation to the hardware accelerator proposed in this thesis in Chapter 4, called Nexus++, and can be configured to either offload tasks' management to Nexus++, or use its own (software) structures. VSs enabled evaluating Nexus++ with real applications.

The hardware accelerator presented in this thesis is based on Nexus [109]: a hardware accelerator that is restricted to the Cell BE processor [26]. Nexus has limitations on the number of parameters a task can have and on the number of tasks that can depend on a certain memory segment. The work presented in this thesis solves the limitations present in Nexus, in addition to providing a more optimized solution for task graph management as shown in later chapters.

1.9 Contributions

As discussed before, dependency-aware task-based programming seems to be the most promising programming model at the moment as it provides a transparent solution for programming multicore systems. Nevertheless, its scalability is limited by the runtime overhead, mostly noticeable when an application has fine-grain tasks, complex dependency patterns, or both.

Moreover, the runtime system shares resources with the worker threads that run the user application. This introduces potential side effects such as cache polluting with task graph data and other runtime-system structures and will lead to replacing the worker threads' data in the cache memory. The problem of resource sharing becomes worse for the worker threads, if there is more than one runtime thread, which usually have higher priority.

The main contribution of this thesis is to offload the heaviest part of the runtime system - task graph management - to a dedicated hardware accelerator, in order to accelerate the runtime system, as well as to save some conflicts on using the shared resources (microprocessor cores and memory system) by the runtime system and the user applications. Moreover, a high-level application programming interface (API) is presented which enables system programmers of utilizing the proposed hardware task graph manager to accelerate their own runtime libraries.

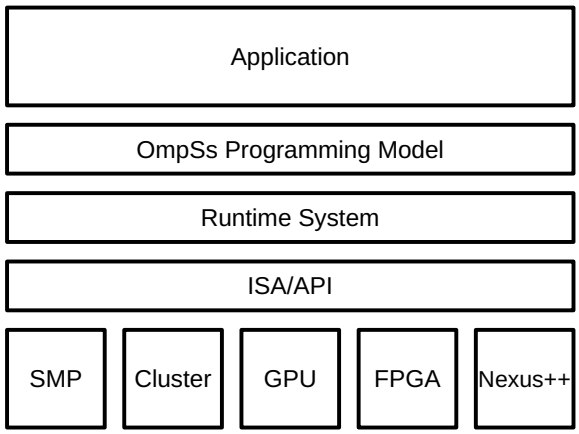


Figure 1.13: A high-level picture the computation model of OmpSs shown in Figure 1.7(a) with the proposed hardware accelerator.

The trade-off is the extra communication between the runtime system and the dedicated hardware module, and the eventual cost of the hardware module itself in the form of logic gates and power consumption.

In more details, the list of contributions includes:

1. a configurable hardware accelerator for task graph management is presented,
2. support for fine-grain tasks, relieving the programmer from the burden of tasks grouping,
3. support for tasks with arbitrary number of arguments,
4. support for dependency-lists of arbitrary size,
5. enabling discovery of distant parallelism, using large window of in-flight tasks.
6. a high-level software API to enable integration of the hardware accelerator with any task-based runtime library.

The proposed hardware accelerator, named Nexus++ in its first prototype and Nexus# in its latest, has been extensively evaluated using different benchmarks of various granularity and dependency patterns. Nexus++ and Nexus# show an improved application scalability when compared to the OmpSs software runtime system (called Nanos++).

Adding Nexus++ (or Nexus#) to the high-level computation model of OmpSs in Figure 1.7(a), results in the system shown in Figure 1.13, where the same level of abstraction is preserved, but this time using a lighter runtime system.

1.10 Scientific Papers

This work has contributed to several publications in international conferences and workshops, upon which this thesis is based. These are (listed in chronological order):

1. Tamer Dallou, Ben Juurlink, and Cor Meenderinck. *Improving the Scalability and Capabilities of the Nexus Hardware Task Management System.* First International Workshop on Future Architectural Support for Parallel Programming (in Conjunction with ISCA'11, 2011).
2. Tamer Dallou and Ben Juurlink. *Hardware-Based Task Dependency Resolution for the StarSs Programming Model.* 41st International Conference on Parallel Processing Workshops (ICPPW), SRMPDS, 2012.
3. Tamer Dallou, Ahmed Elhossini, and Ben Juurlink. *FPGA-Based Prototype of Nexus++ Task Manager.* 6th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS) 2013, Co-located with SC 2013, 2013.
4. Nina Engelhardt, Tamer Dallou, Ahmed Elhossini, and Ben Juurlink. *An Integrated Hardware-Software Approach to Task Graph Management.* 16th IEEE International Conference on High Performance and Communications HPCC-2014, 2014.
5. Tamer Dallou, Nina Engelhardt, Ahmed Elhossini, and Ben Juurlink. *Nexus#: A Distributed Hardware Task Manager for Task-Based Programming Models.* The 29th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015.

1.11 Thesis Organization

This chapter introduces the scalability problem in the era of multicore processors. It sheds light on the challenges of parallel programming and highlights the benefit of dependency-aware task-based programming as a promising solution to the parallel programming challenges of multicore systems. It motivates for this thesis work by highlighting the overhead resulting from the runtime systems of such a programming model, which is a central bottleneck that limits their scalability. The remainder of this chapter lists the contributions of this thesis, the scientific papers published in international conferences and workshops upon which this thesis is based.

Chapter 2 elaborates in more details on OmpSs as a relevant task-based programming model, in addition to special case of parallelizing H.264 video decoding using OmpSs. The remainder of this chapter gives a summary of the related work.

Chapter 3 introduces the architecture of *Nexus++*: a hardware accelerator for task-based programming models in general, and for the StarSs programing model in particular. It describes the design as a SystemC model, and evaluate it using several synthetic benchmarks as a preliminary design-space exploration of the real hardware presented in the next chapters.

In Chapter 4, an improved architecture of Nexus++ is presented as a VHDL prototype, along with an in-house light-weight runtime system capable of running several benchmark with Nexus++ handling task-graph management. Part of evaluating Nexus++ in this chapter, Nexus++ bitstream (hardware) was downloaded to an FPGA, and a special interface was designed to communicate with the host multicore machine over the PCIe bus. Several real benchmarks were used for evaluation, as well as their traces to do further simulations for large number of cores that do not exist in the multicore machine used in the evaluation.

Chapter 5 analyzes the bottlenecks in the architecture of Nexus++, and solves them by introducing a distributed task-graph architecture called the Nexus#. The latter architecture pushes the scalability of applications with fine-grain tasks further. It was evaluated using trace-based simulations of the VHDL prototype.

Chapter 6 suggests future prospectives to further develop this research work and finally Chapter 7 draws conclusions.

2 Background

2.1 The StarSs and OmpSs Programming Models

The StarSs programming model is a task-based dataflow model that creates tasks out of the user program, which can potentially run in parallel. The *StarSs* name is composed of the wildcard character (*) and the *Superscalar* abbreviation *Ss*, as it exploits task-level parallelism (TLP) based on data dependences between tasks to the same way instruction-level parallelism (ILP) is exploited in out-of-order superscalar processors. StarSs is meant to target several architectures, for example CellSs for the Cell architecture [19, 26], SmpSs for symmetric multiprocessors [123], etc.

In contrast to other task-based programming models, StarSs is dependency-aware, meaning that the programmer can annotate the source code with certain *pragmas* indicating the inputs and outputs of tasks. Using the information provided by the *pragmas*, the StarSs runtime system tracks the inputs/outputs dependences between tasks and builds at runtime a task graph to decide which tasks are ready to run at any point in time. The runtime system also tracks tasks' completion and updates the task graph accordingly, in order to move any dependent tasks to the pool of ready tasks. This way, the result of the parallel execution of a certain application is equivalent to its sequential counterpart. StarSs also enables the programmer to optimize his/her code, by bundling multiple tasks together in order to decrease the per-task management overhead, or to exploit data locality by bundling a chain of dependent tasks.

OmpSs is a programming model [28, 54] that brings the benefits of StarSs and OpenMP in one model. It adapts the task-based execution model and enables the programmer to run his/her code on a heterogeneous machine, by specifying the target of each task: SMP, cluster, GPU, or FPGA. Its high-level model can be seen in Chapter 1, Figure 1.7(a). It decouples the hardware from the application, with the

		1st task		
		input	output	in/out
2nd Task	input	none	**true**	**true**
	output	anti	output	output
	in/out	anti	**true**	**true**

Table 2.1: Depedences between tasks accessing common objects.

runtime system orchestrating the usage of the hardware resources and transparently managing parallelism for the programmer.

OmpSs enables uncomplicated exploitation of task-level parallelism. It provides the programmers with *pragmas* to identify code blocks that can potentially run in parallel. OmpSs extends the task model to capture data dependences between tasks. Using the syntax `#pragma omp task in(A,...) inout(B,...) out(C,...)` (with `A`, `B`, `C` are the objects passed as operands to the task), the programmer can indicate which inputs a task consumes and which outputs it produces. An *Input* operand indicates a read-only operation, whereas an *output* argument indicates a write-first operation meaning that this operand will be written first and might be read and written later on, therefore the initial value is irrelevant. An *in/out* operand indicates that the task may read and write this operand and that its initial value might be read. In 2013, OpenMP also introduced OmpSs-like task dependences in version 4.0 of the programming model[23], using the *depend(dependence-type: list)* directive. For example, a task that reads memory locations *A and B* and writes memory locations *B and C* is annotated by `#pragma omp task depend(in:A) depend(inout:B) depend(out:C)`.

The arising dependences between two tasks accessing a common object can be seen in Table 2.1 based on their access mode to the common object. The table shows an example of two tasks marked as *1st task* and *2nd task* indicating their order in the program, which is the order at which they are to be inserted in the task graph. A *none* dependence means that no dependence exist between the two tasks and therefore they can run in parallel. A *true* dependence on the other hand means that the 2nd task needs some data which is produced by the 1st task and therefore depends on it. This type of dependence is also known as *Read After Write (RAW)* dependence. There are two other types of dependences in the table, namely the *anti* and the *output* dependences which are analogous to data hazards in CPU's pipelines [72, 122].

Anti and *output* dependences, also known as *Write After Read (WAR)* and *Write After Write (WAW)* dependences respectively, are not true dependences since no data flow between the two tasks in the form of a producer-consumer relationship. However, they arise when two or more tasks use the same memory location to write their data. Therefore such dependences can be removed by using different memory locations by the tasks, a technique known as renaming. Renaming of function operands can be done by the runtime system in order to remove false dependences and to build the task graph solely based on true dependences between tasks. Although this might simplify the resulting task graph, it increases the runtime overhead, since the runtime should keep track of multiple versions of the same object and should ensure that a certain task accesses the appropriate version of the object.

OmpSs is based on *mercurium*, a source-to-source compiler that transforms the pragmas in the user code into function calls to its runtime library called *Nanos*. *Mercurium* translates task annotations to Nanos runtime library calls, so that whenever an annotated function is called, rather than being directly executed a task will be generated and added to the task graph based on its input/output parameters. The resulting code can then be compiled by a conventional source-to-object compiler such as gcc. When running the resulting executable file, the runtime system builds the task graph by spawning tasks one after the other according to their position in the program. A dependence between two tasks means that the tasks must be executed in the order they were generated in the first place. The resulting task graph is a directed acyclic graph (DAG), since tasks can only depend on other earlier tasks in the program order. The root nodes in a task graph are the tasks that are executing or ready to execute. When they finish executing, the runtime system removes them from the task graph, resulting in new root node(s) that can be scheduled to run.

OmpSs implements a thread-pool model where the master thread starts execution and generates tasks, which can be executed using the other threads in the pool. By dynamically maintaining the task graph, the runtime system resolves dependences between tasks and decides which task is ready to run at any point in time.

The programmer does not need to care about synchronization between the tasks, as this is done implicitly by the runtime system. OmpSs can be used also in heterogeneous, multicore systems. Such systems may include SMPs, GPUs, FPGAs, etc. This can be done by providing alternative implementations for a certain function for

different devices. The runtime system is then free to exploit parallelism in the user code as much as it can.

The StarSs and OmpSs Programming Models have been developed and used in several European research projects such as the SARC project (Scalable computer ARChitecture, www.sarc-ip.org) [132, 135], the ENCORE Project (ENabling technologies for a programmable many-CORE, www.encore-project.eu), the Mont-Blanc projects (www.montblanc-project.eu) [133], the TERAFLUX project (www.teraflux.eu) [65], the DEEP project (www.deep-project.eu) and the DEEP-ER project(www.deep-er.eu), the INTERTWinE project (www.intertwine-project.eu/) and the AXIOM project (Agile, eXtensible, fast I/O Module, www.axiom-project.eu) [152].

2.2 H.264 Video Decoding in OmpSs

H.264 [165] is a video standard widely used in consumer devices worldwide. Andersch et al. in their work presented the case of parallelizing H.264 using OmpSs [7]. They demonstrated how OmpSs can be used to exploit pipeline parallelism in H.264 decoding and the relatively ease of programmability of OmpSs in contrast to Pthreads parallelism.

```
1 MB_type* X[NB_WIDTH][NB_HEIGHT];
2 //MB_type: a data str. that rep. MB dependencies.
3 #pragma omp task input(left, upright) inout(this)
4 void decode(MB_type* left, MB_type* upright, MB_type* this){...}
5 void main(){
6  int i, j;
7  init_matrix(X) ;
8  for(i=0; i<NB_WIDTH; i++)
9   for(j=0; j<NB_HEIGHT; j++)
10    decode(X[i][j-1], X[i-1][j+1], X[i][j]);
11  #pragma omp taskwait
12 }
```

Listing 2.1: OmpSs example of macroblock wavefront decoding in H.264

Listing 2.1 shows an OmpSs example of exploiting parallelism using pragmas for macroblock decoding in H.264 [7]. Each frame in an H.264 sequence is divided into several macroblocks and each macroblock covers 16×16 pixels. In the example shown

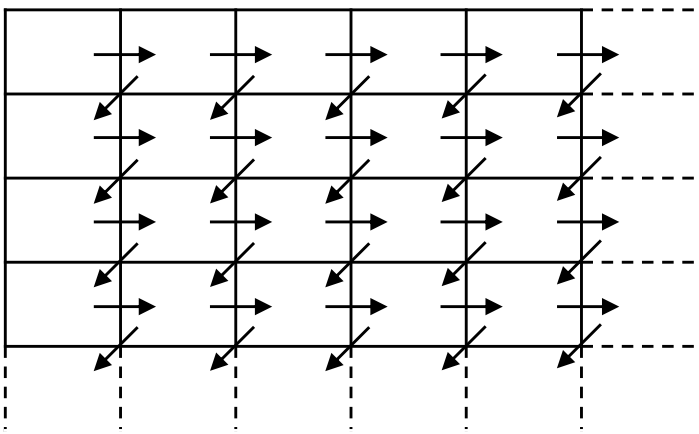


Figure 2.1: The dependency pattern in the macroblock reconstruction stage in H.264 video decoding. Reconstructing a certain macroblock requires some pixel areas from adjacent macroblocks.

in Listing 2.1, the function *decode()* is called inside a nested loop, iterating over the macroblocks in a video frame.

Calculating the *decode()* function for a certain macroblock requires the results of the *decode()* function on the left macroblock and the three macroblocks above it [156]. This dependency pattern of H.264 macroblock decoding is shown in Figure 2.1, simplified by having each macroblock being dependent only on the macroblocks at its left and upper-right, since the latter macroblock depends on its left macroblock. This dependence pattern appears repeatedly in H.264 macroblock decoding, however since H.264 video decoding is data-dependent and thus has variable execution times, it is hard to apply static task scheduling. Therefore, H.264 video decoding is amenable to dynamic task scheduling to improve decoding performance. Exhibiting variable task execution times, irregular parallelism, and fine grain tasks, in addition to being widely used in academia and industry make H.264 video decoding a representative and driving application for this thesis.

When a function declaration is annotated with the *omp task* pragma, any calls to the function are turned into task submissions. The input and output parameters of the task should be specified in the annotation pragma as well as shown in Listing 2.1. This permits the runtime system to detect dependences between tasks and launch them only when all their input data are available. In the example of Listing 2.1, every time the function *decode()* is called, a task is generated. The call returns immediately,

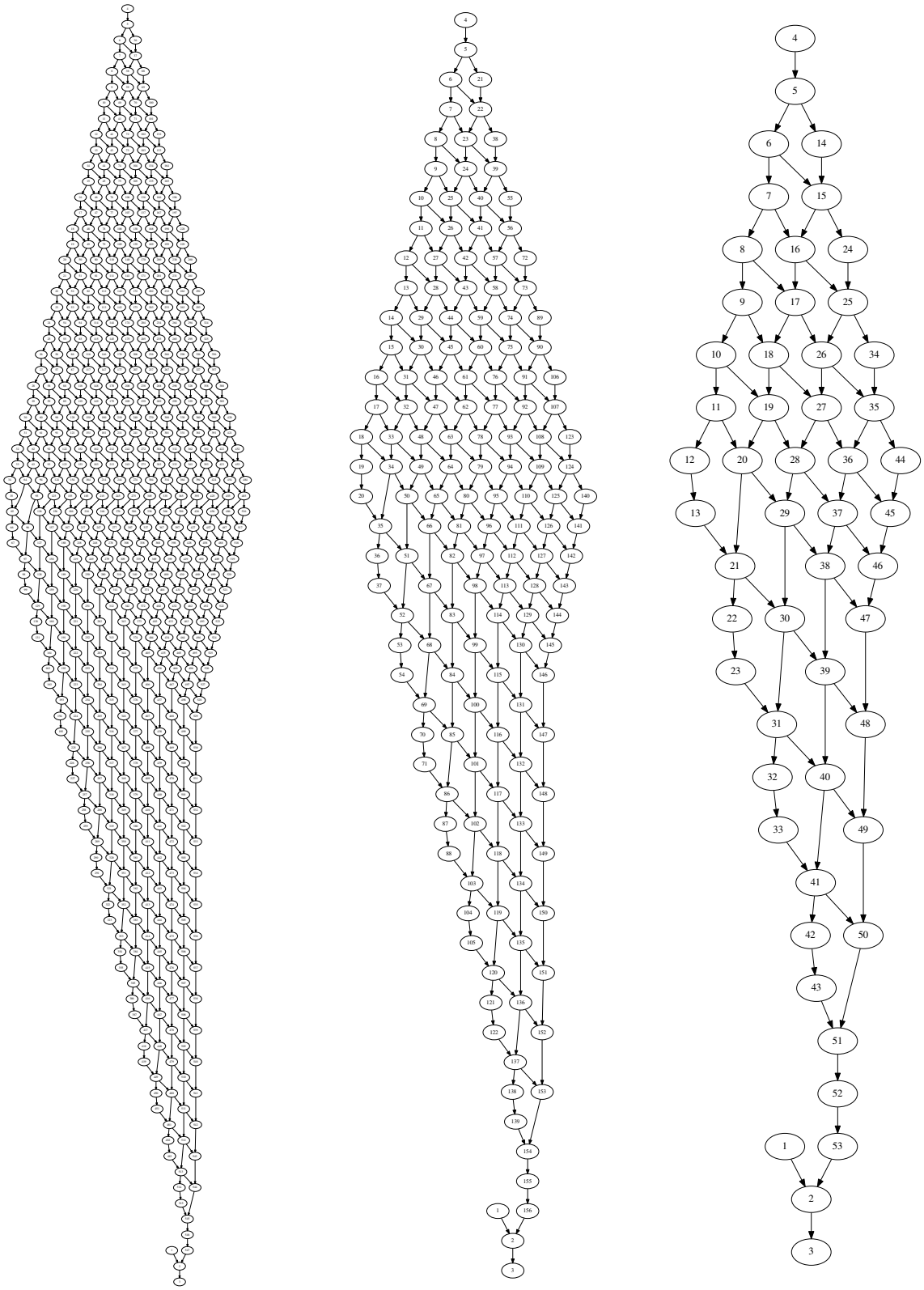
allowing the submission of more tasks concurrently to their execution. At the end, a *taskwait* pragma makes the thread wait for completion of the submitted tasks.

Having identified the tasks and the direction of their parameters, the OmpSs environment builds the task graph at runtime and the task-level parallelism is detected and exploited. Ready tasks are scheduled to run on the worker cores and once they finish, the task graph is revisited to release dependent tasks.

Central to this thesis is the work by Andersch et al. [7], where the authors show the need to group the processing of several macroblocks to overcome the task management overhead of the OmpSs runtime system, which constitutes a bottleneck limiting the scalability of applications in OmpSs. In the same work, the authors show how grouping several tasks together is not a trivial task and requires the programmer to interfere to preserve the correct execution order of tasks, thus increasing code complexity.

Figure 2.2 shows different task graphs when running the H264dec benchmark for one FHD video frame, which were generated using *TEMANEJO*, a tool that enables the visualization of task-dependency in task-based programming models [24]. A node represents one task and the edges between nodes are the dependences between tasks. A FHD video frame in H.264 has 8100 macroblocks, each of which is 16×16 pixels. Reconstruction one macroblock alone is a fine grain task. The figure shows three task graphs for macroblock reconstruction in H.264. They differ in the number of macroblocks grouped per task in order to coarsen the tasks and minimize the runtime overhead. It can be seen how the task graphs build up with respect to time (top-to-bottom). Tasks at the same row in the graph are independent and given enough resources (worker cores) they can be run in parallel. Comparing the graphs side by side shows how the number of parallel tasks is reduced as more macroblocks are grouped together and assigned to one task.

The number of macroblocks assigned to each task is defined for the main body of the video frame and is not the same for all tasks. This is due to the fact that macroblocks on the edges of the frame require extra care from the programmer to preserve the overall order of macroblocks reconstruction. In the example of Figure 2.2(c) where 256 macroblocks are grouped together per task a FHD frame should be processed by 32 (=8100/256) tasks, the figure has 53 nodes.



(a) 16 MBPT (b) 64 MBPT (c) 256 MBPT

Figure 2.2: Different task graphs of H.264 video decoding one FHD video frame varying the number of macroblocks to be reconstructed per task (MBPT). Graphs were generated using *TEMANEJO* [24].

2.3 Related Work

Parallel programming has always been an active field, even before the breakthrough of multi/many core processors. Nowadays, it receives more importance as the target has shifted from programming supercomputers to normal consumer devices.

The quest to provide performance with less power can be achieved by further dividing a certain workload into smaller pieces, by extracting any potential parallelism in it and mapping them to the worker cores. This way, the worker cores can be clocked at a lower frequency, hence consuming less power and still meeting the requirements of the workload.

The goal of this research work is to improve the scalability of applications on systems with large number of cores, by exploiting fine-grain parallelism in applications while at the same time preserving the ease of programmability. Therefore, I'll mention the related work, to the best of my knowledge, that enables programmability of multicore systems with minimum or no interference from the programmer, as well as those related to study of the runtime overhead of managing parallelism. In addition I discuss similar approaches of hardware acceleration of parallelism detection and management.

2.3.1 Software-Based Approaches

In the literature there are several programming models that aim at improving the parallelism and scalability of the user applications. Most of them, however, assume independent tasks and are optimized for a certain application, a certain platform, or both.

As previously introduced, OpenMP [14, 23], the StarSs family of programming models [18, 123, 127, 150] including OmpSs [27, 54], and StarPU [12] are good examples of a high-level parallel programming model, which require the programmer to only annotate sections of code that can potentially run in parallel. The runtime system then takes care of maintaining dependences between tasks and scheduling those ready to run. To achieve that the runtime system can introduce overheads which can limit the scalability to large number of cores.

Nanos++ [37] is a runtime system that handles task management for OmpSs and OpenMP applications. It is developed and maintained at the Barcelona Supercom-

puting Center. Nanos++ is a feature-rich system that supports running tasks on GPUs and clusters, in addition to symmetric multiprocessors. It supports also different scheduling policies and rich instrumentation capabilities. Nanos++ is the baseline system used for performance evaluation in this thesis.

VSs [55] and MTSP [118] are other two runtime systems for OmpSs and OpenMP task-based programming models. Targeting high performance, they are developed as light-weight libraries that support basic task directives (task dependencies and synchronization barriers), with limited instrumentation features compared to Nanos++.

Gautier et al. [63, 64] presented X-kaapi, a multi paradigm runtime system for multicore architectures including the dataflow paradigm, where the user can specify tasks using the *#pragma kaapi task* to declare tasks with their access modes. The X-kaapi runtime system handles task creation and synchronization.

Lyberis et al. [61, 104, 105] presented Myrmics, a task-based runtime system targeting non cache-coherent, heterogeneous manycore processors. It exploits the producer-consumer property using a dataflow model to provide software-based cache coherency on non cache-coherent architectures, by transferring data from producer to consumer CPU cores using decentralized runtime agents. It uses the same principle to prefetch data into the cache memory of the consumer CPU in cache-coherent, shared memory architectures. Myrmics has been implemented and evaluated on a specifically designed heterogeneous 520-core FPGA prototype, modeled after common trends in manycore processors design. The proposed hardware task manager in this thesis supports double-buffering in a similar way as Myrmics to overlap computation and communication and therefore enhances performance. Although not implemented with a heterogeneous architecture yet, integrating our hardware task manager with Myrmics can be an interesting approach towards Exascale architectures.

Wang et al. [162] proposed FPL (Flexible Programming Model) and the MP Tomasulo [163], where they developed a runtime system for StarSs applications, targeting FPGA-based embedded microprocessors (such as Microblaze, PowerPC, or ARM cores) and custom IP cores, which are responsible for accelerating specific tasks in hardware. Their runtime system supports renaming of tasks' parameters to overcome false (WAW and WAR) inter-task dependences.

Pop and Cohen [129] developed another dataflow extension to OpenMP called OpenStream, where the programmers can specify dependences between tasks using *streams*.

This work has been extended by Drebes et al. [53] to support non-uniform memory access (NUMA) architectures, by developing a NUMA-aware task and data placement algorithm to achieve high data locality and at the same time preserve a uniform abstraction of the hardware resources by the programing model.

The literature has some articles about similar task dataflow programming models that support task dependencies [2, 40, 70]. Agrawal et al. [2] developed the *NABBIT* library for Cilk++, which enables the programmers to specify task inputs and outputs, and therefore build a task graph to exploit parallelism.

Vandierendonck et al. [158] also extended Cilk++ by adding dependency clauses (input, output or both (inout)) on task arguments to facilitate complex patterns of parallelism such as pipeline parallelism, without increasing code complexity or loosing performance. The authors also presented a unified scheduler [159] for task dataflow applications, that enables the programmer to use algorithms with dependency-aware tasks and algorithms with classical fork-join style.

Gupta et al. [70] in their approach exploit function level parallelism and by mapping function calls to tasks, they construct task graphs based on the inputs and outputs of the functions in applications. Although Gupta et al. provide the programmer with a clear API to specify functions that should be considered for dataflow execution, the needed effort is relatively more cumbersome when compared to directive-based approaches, such as OmpSs and OpenMP.

Loghin et al. [102] worked on optimizing a fine-grain dataflow graph in order to efficiently run it on a multicore processor. They did this by coarsening tasks by applying node fusion on both loops and functions. Using matrix multiplication and prime numbers counting as benchmarks, the authors emphasized the importance of task granularity and its effect on application scalability.

Other approaches in the literature that utilize dataflow task graphs to exploit parallelism include Data-Driven Tasks [149], where the programmer can use the *await* clause to specify the arguments a task requires before it can start. A *put()* operation can be called by some task to provide an argument in the *await* clause of another task.

The OSCAR (Optimally SCheduled Advanced multiprocessoR) Multigrain Parallelizing Compiler [82, 84, 88, 112, 117], the MAPS (MPSoC Application Programming Studio) framework [34, 35, 97], Automatic Parallelization tool [60] and the DiscoPoP

(Discovery of Potential Parallelism) project [48, 98], where static and dynamic analyses are performed to automatically extract parallelism from the sequential programs.

Tareador [15] is a tool that helps the programmer iteratively annotate the sequential program, by providing a simple API and a graphical representation of the dataflow in the user program, identifying potential task-based parallelism in the sequential programs. Other performance analysis, debugging, and visualization of dataflow task programming models include Paraver [36, 38], Aftermath [51, 52], DAGvis [77], and TEMANEJO [24, 25, 74]. The latter tool has been used in this thesis, in particular for generating the task graphs of some of the benchmarks used for performance evaluation in this thesis.

Vandierendonck et al. [160] analyzed the runtime overhead for task-based programming models, identifying two concepts that characterize a task graph: 1- *versions*, capturing renaming of operands, and 2- *generations*, capturing sets of parallel tasks. The authors analyzed the task graphs of applications programmed with task-based models and using renaming of operands, they claim a reduced dynamic runtime overhead for maintaining the task graphs by removing the *anti* and *output* dependencies shown previously in Table 2.1. Moreover, they emphasized that grouping of parallel tasks into generations allows a later dependent task to wait for a generation of parallel tasks as a whole, rather than adding it to the waiting lists of the individual tasks in the generation, simplifying dependence tracking and minimizing runtime overhead while generating hyper task graph structure. Consequently, the authors proposed a software approach for task management that exploits both versions and generations in a task graph, and stored the task graph in a linked list structure reducing the number of edges in the task graph. They proposed an edge-less scheme as well, where tickets are issued to tasks upon task spawning and by maintaining counters, tasks with ticket value equal to the global counter are ready to run.

Using microbenchmarks, the overhead of their proposed software task graph manager can go as low as 400 cycles (0.2 μs on their test machine, a 48-core AMD Opteron 6172 at 2.1GHz) per task, for the case of a microbenchmark with all independent tasks, i.e., tasks are ready to run immediately after spawning. In their analysis, however, the authors assumed that any task has one operand only, claiming that their findings are applicable to tasks with multiple operands. Although this might be true for the case of adding dependent tasks to the different waiting lists of the output operands of a

certain task, the runtime overhead estimation can be more complicated when having a certain task (with multiple input operands) on the waiting lists of multiple tasks. It is also not clear if the renaming overhead is taken into account or not.

Using real benchmarks with complex dependency patterns on the other hand, such as Cholesky factorization (described in the previous chapter in Section 1.6), Vandierendonck et al. showed that the different task graph management schemes exhibit nearly the same scalability behavior. The task graph management overhead is dominant at smaller task sizes, with the ticket-scheme being about 10% faster than the other edge-centric schemes.

Although renaming of operands is not handled by the proposed hardware task manager in this thesis, a generation-like mechanism is implemented, by handling a set of parallel tasks accessing a certain object as a group, allowing a potential dependent task (a task that writes this memory object later on) to wait for the whole group and not the individual tasks. Therefore, the hardware task manager presented in this thesis can be classified under the hypergraph paradigm discussed by Vandierendonck et al. [160].

TurboBŁYSK [128] presented a framework for OpenMP 4.0 [23] to support fine grain tasks. The authors proposed a scheme that re-uses previously resolved dependency patterns to minimize the runtime overhead. The proposed pattern-saving mechanism is based on programmer annotations, where the programmers can use the *dep_pattern* clause introduced in the scheme. TurboBŁYSK task graph is similar to the hypergraph paradigm presented in Vandierendonck et al. [160] and has shown significant scalability improvement over OmpSs.

2.3.2 Hardware-Based Approaches

To the best of my knowledge, most of the contributions in the literature that propose hardware support for task parallelism focus on the process of scheduling existing tasks without participating in the process of dynamic dependency resolution. Among these is Carbon [91], a hardware accelerator for dynamic task scheduling, with hardware distributed task queues for low-latency retrieval of tasks and with task stealing support. In addition to the low-latency queues, the Carbon architecture has a per-core prefetch unit to hide the latency of accessing the queues. A software API is provided to enable the programmers to create processes that enqueue and dequeue tasks from Carbon queues. Carbon's software library implements an overflow mechanism, in case that the number of in-flight tasks exceeds the queues limited capacity, which pushes the extra tasks to the memory system back and forth, thus supporting virtually an unlimited number of tasks and processes. Several benchmarks with loop-level and task-level parallelism were used to evaluate Carbon's performance on a cycle-accurate, execution-driven multicore processor simulator developed by the authors themselves [91]. Carbon showed to outperform software implementations of dynamic task schedulers.

The prefetching mechanism implemented in Carbon is similar to the double-buffering mechanism implemented in this thesis, as described in Chapter 3. Although Carbon provides a low-latency mechanism to enqueue/dequeue ready tasks to/from the queues, the programmer is responsible for handling dependences between tasks and pushing only ready tasks into the queues. Carbon-like queues can be integrated with the proposed hardware task manager in this thesis in order to provide several ready tasks queues with task stealing support, rather than using one centralized ready tasks queue, which will be eventually a bottleneck in the case of a large number of cores.

Hoogerbrugge and Terechko [75] proposed a task scheduling unit based on Carbon. The authors evaluated their system using a simulated architecture, where they integrated their proposed hardware task scheduling unit with a TriMedia-based multicore system and a hardwired H.264 entropy decoding unit. As in Carbon, parallelism extraction and management were left to the programmer. Nevertheless, the authors demonstrated good scalability results (up to 14×) for the case of decoding an UHD H.264 video file on a simulated 16-core prototype.

Sanchez et al. [138] introduced an asynchronous direct messages (ADM) unit, which can be integrated with individual cores in a multicore system. The unit enables, at low overhead, direct exchange of short messages asynchronously between threads without going through the memory hierarchy. It can be used to implement low-overhead runtime systems and schedulers that can exploit fine-grain tasks without being a scalability bottleneck. In their experimental setup, the authors simulated a tiled, large scale (up to 128 cores) multicore system. They evaluated their approach by comparing it to a software-only scheduler and to a Carbon-based model. Their proposed ADM-based task scheduler outperformed software-only schedulers and matched the performance of the Carbon-based model.

Castrillón et al. [32, 33] proposed *OSIP* (Operating System Instruction-set Processor); a programmable Application Specific Instruction set Processor (ASIP) for accelerating the scheduling, mapping and synchronization tasks in MPSoCs. While traditional ASIP designs target a certain application or an application domain [79], OSIP's target is to accelerate the operating system's services (OSIP's name OS Instruction-set Processor), making it suitable to build efficient and scalable applications or system libraries. The authors achieved this by profiling commonly used scheduling algorithms (round-robin, priority-based, first-come first-serve, fair queue) in the hierarchical scheduling [66] approach for heterogeneous architectures, and designed the OSIP architecture accordingly. In particular, OSIP provides hardware primitives for efficient memory access, control and arithmetic operations used heavily in scheduling algorithms. Moreover, OSIP provides a register interface which attaches to a MPSoC through the latter's standard interconnect, in addition to utilizing the MPSoC's interrupt lines to trigger the worker cores when needed. The authors also presented a set of well-defined programming APIs for task management including *CreateTask*, *SuspendTask*, *DeleteTask*, in addition to other synchronization APIs. Using those APIs, the authors implemented a parallel H.264 video decoding exploiting macroblock-level parallelism [110].

Integrating OSIP with an ARM9-based [8] multicore system in a virtual prototype, OSIP demonstrated significant improvements over software solutions. The simulated system consisted of variable number of ARM9 (2 to 15) cores, with different configurations of OSIP and an idealized interconnect. All of the ARM9 cores in addition to the OSIP core were clocked at 200 MHz. For a 15-processor configuration, OSIP was able to manage tasks as small as 25 kcycle. On an 8-processor configuration, OSIP

was able to manage even finer-grain tasks of size 10 kcycle. This is comparable with Nexus# which is able to manage fine-grain tasks of size 12 kcycles ($4.6\mu s$ on a Xeon E7-4870 multicore machine, running at 2.40GHz, shown in Table 1.3), with complex inter-task dependences and scaled up to 6 cores as shown in Chapter 5.

Zhang et al. [176] analyzed the integration of OSIP in shared and distributed memory multicore systems. The authors emphasized on the importance of the communication infrastructure in order to fully utilize OSIP. This is the case for any runtime hardware coprocessor including Nexus++ and Nexus#, where high-speed communication is critical for system performance, since communication takes place very often and in small packets.

Hardware support for OS services such as task scheduling or even moving OS functionality to dedicated hardware structures is an active topic in the literature. The hardware OS kernel (HOSK) presented by Park et al. [121] provides hardware primitives to reduce the multithreading overhead in RISC-based multiprocessors. In order to reduce the overhead of context switching, HOSK proposed local context controllers to be embedded in every RISC processor that communicate with HOSK's central context manager over a dedicated bus to prefetch the next ready task. Although HOSK showed reduced multithreading overhead for the presented RISC-based environment, the authors neither presented a software API to exploit HOSK nor did they give insights on how to integrate HOSK with generic multiprocessors.

The SystemWeaver presented by Lippertt [100] uses task-based parallelism and provides hardware primitives for fast task creation, synchronization and scheduling in heterogeneous multicore architectures.

The Swarm architecture [83] is another architecture which integrates distributed hardware task queues in a tiled, cache-coherent chip multiprocessor. The authors depend on the programmer to specify the order at which tasks can be executed by assigning timestamps to tasks upon tasks creation. Their task execution model uses thread-level speculation, in which tasks are executed speculatively out-of-order and are concluded in-order. To achieve high scalability, they exploit fine-grain tasks and use the distributed hardware queues for reducing task management overhead and for supporting a large speculation window. Using an event-driven simulation model, they achieved $3 - 18\times$ speedup on a 64-core chip multiprocessor, compared to the

software-only parallel implementation of a set of benchmarks from the domains of graph analytics, simulation, and databases.

Other modern architectures adapting task-based programming with hardware support for fine-grain tasks vary from massively parallel architectures such as the Anton 2 [69, 142] supercomputer used for molecular dynamics simulations, to embedded systems such as the hardware thread scheduling unit for data-driven multithreading [92], presented by Matheou and Evripidou [107]. Although those systems require the dependency graph to be provided by the programmer and focus on optimizing thread/-task management, they share the general approach with the hardware task manager proposed in this thesis of providing hardware support for the software side.

In the industry, Texas Instruments presented the Keystone II architecture [21, 81] that includes the Multicore Navigator, which utilizes hardware queues for fast task dispatching and scheduling.

Few works exist that target dynamic dependency resolution in task-based programming models. Most of them are either optimized for a certain application, a certain platform, or both.

For example, a look-ahead task management unit (TMU) optimized for H.264 video decoding is presented by Själander et al. [143]. Based on the dependency pattern of H.264 video decoding shown in Figure 2.1, the authors designed a simple TMU that monitors which core decodes which macroblock in a video frame, and looks ahead in time to prepare the dependent tasks of that macroblock so that the core can immediately decode them once it finishes decoding its current macroblock. Although this approach is beneficial for H.264 video decoding, it is not a generic solution, and even for parallelization opportunities in H.264 other than macroblock decoding, for example pipeline parallelism or inter-frame parallelism [7].

Al-Kadi et al. [3] proposed a hardware task scheduler (HTS) optimized for applications that exhibit repetitive inter-task dependency patterns with focus on H.264 video decoding. However, it requires the programmer to isolate the parallel kernel in the source code and to identify the repetitive dependency pattern between tasks. The proposed approach in this thesis on the other hand provides hardware acceleration for the runtime systems of StarSs/OmpSs and similar programming models to reduce their overhead, leaving it to such programming models to provide a high level abstraction

that facilitates programmability and performance portability as introduced in the first chapter, thus supporting irregular parallelism in different application domains.

Seidel in his Phd dissertation [139] proposed a task-level programmable processor that utilizes dataflow task-based parallelism and integrates a hardware unit that manages dependencies between tasks. Limberg et al. [9, 99] proposed the CoreManager, a hardware unit based on Seidel's work [139]. The CoreManager is responsible for detecting task dependencies, dynamically maintaining the task graph and for task scheduling. It also handles data transfers between the different cores in the system. A task-based programming model similar to OmpSs is also presented, which enables the programmers to define tasks along the target architecture using simple annotations. Furthermore, the programmer is required to specify the inputs and outputs of each task at the point of task instantiation. The CoreManager is then integrated in a heterogeneous multicore architecture called the Tomahawk MPSoC, which is a software defined radio platform with support for multimedia applications, demonstrating the efficiency of the CoreManager managing parallelism in heterogeneous architectures.

Another approach similar to the one proposed in this thesis is the Task Superscalar presented by Etsion et al. [56–58], who proposed a hardware task management unit for the StarSs runtime system, based on the similarity between task-level parallelism in task-based programming models and instruction-level parallelism in out-of-order superscalar processors. In their work, the authors assume that a task in the StarSs programming model is the basic unit of computation that can run on a core in a heterogeneous multicore processor. Their proposed accelerator tracks dependences between tasks in the same way dependences are tracked between instructions in a modern superscalar processor. The tasks are added to task reservation stations, where they wait for their inputs to be produced by previous tasks, before they can be executed on one of the processor cores. A VHDL prototype is presented for it in [174], but it is only evaluated using high-level simulations. As shown in Chapter 5, the hardware implementation, compared to ours, is relatively expensive. PICOS [148, 175] is also a hardware implementation of the task superscalar [56] to support the OmpSs programming model, which improves the hardware utilization and latencies over the original task superscalar design. The task superscalar architecture supports renaming of tasks' parameters, a feature that is not supported by our hardware task manager until now.

Meenderinck et al. [109] proposed Nexus (the baseline of our design), a hardware task manager that is restricted to the Cell BE processor [26]. Moreover, Nexus has limitations on the number of parameters a task can have and on the number of tasks that can depend on a certain memory segment. The work presented in this thesis solves the limitations present in Nexus, in addition to providing a more optimized solution for task graph management as shown in later chapters.

Finally, based on Nexus++, LG Electronics presented Tioga [46]: a hardware task manager integrated in an ARM-based embedded multicore system, which aims at providing efficient task-graph management to utilize fine-grain task-based parallelism in embedded systems.

3 Hardware-Based Task Dependency Resolution for the StarSs/OmpSs Programming Model

The idea of relieving the programmer from explicitly extracting parallelism and limiting his/her role to light-weight annotating the source code is very promising for contemporary and future multicore SoCs. This approach, introduced by dependency-aware task-based programming models, is what makes it substantially standing out as a key solution for the programmability and scalability problems.

As the multicore processors era and OmpSs (the programming model in focus), are relatively new, a proof-of-concept model for hardware support is thought to be implemented first using SystemC.

SystemC is a C++ class introduced by the Open SystemC Initiative in 1999 [1], which provides the basic primitives to rapidly model hardware modules and concurrent processes, an event-driven simulation platform, in addition to the easy-to-code and debugging capabilities of C++.

This first model, called Nexus++, gives an insight about the feasibility of introducing hardware support for task graph management. Furthermore, it is used to perform design space exploration and get an estimate of the required hardware resources.

3.1 Nexus++ Hardware Task Management System

The multicore system under consideration, shown in Figure 3.1, is assumed to have one *Master Core* that executes the main thread and creates *Task Descriptors*, and several worker cores that execute the tasks.

In StarSs and OmpSs, a task's life cycle starts when a *task* pragma is encountered in the source code.

A *Task Descriptor* contains task-related information such as its function pointer and input/output list. Nexus++ is responsible for the task graph management responsibilities usually carried out by the software runtime system. In an n-core system (one master core and $(n-1)$ worker cores), Nexus++ is composed of n hardware modules:

- one *Task Maestro*, which is mainly responsible for dependency resolution, task scheduling, and load balancing,
- and $n-1$ local *Task Controllers* (TCs), one per worker core, and are mainly responsible for task buffering.

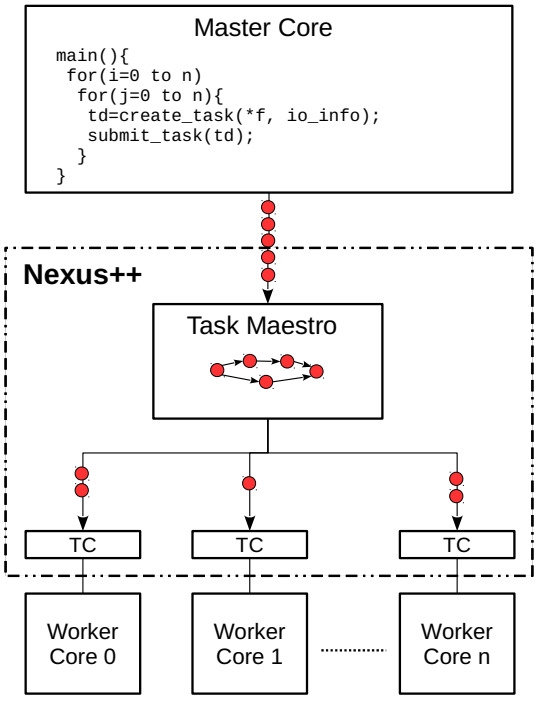


Figure 3.1: Nexus++ in a multicore system.

3.1.1 System Description

A more detailed view of the different components of Nexus++ is shown in Figure 3.2. The main components in the *Task Maestro* include the *Task Pool* where tasks' descriptors are stored, the *Dependence Table* where the task graph is maintained, in addition to several handlers needed to manage the task graph and queues needed for communicating with the multicore system.

This section describes the different components in Nexus++ in the order they get active or accessed, starting from receiving a task from the master core and inserting it to the *Task Pool* and the *Dependence Table*, to sending ready tasks to the worker cores and simulating task's execution, and ending with retiring finished tasks and removing them from Nexus++ structures.

1. **Submitting tasks:**
   When the *Master Core* executes the main program, it generates *Task Descriptors* and sends them to the *Task Maestro* via a dedicated bus; the *TDs Bus1* in Figure 3.2.

   The bus width is chosen to be 64 bit in order to reflect bus widths in concurrent computer architectures, and therefore to give a realistic picture of the performance of Nexus++ if integrated in real multicore systems.

   The *Get TDs* handler; the first component in the *Task Maestro*; is responsible for communicating with the *Master Core* and receives the variable-length *Task Descriptors* (depending on the number of inputs/outputs per task) from the *Master Core* over the *TDs Bus1* and writes them to the *TDs Buffer*. The *Get TDs* handler uses a simple master-slave handshaking protocol using two *request* and *acknowledge* signals.

   The *Get TDs* handler is important so that the *Master Core* is not blocked while the *Task Maestro* is busy processing an earlier submitted task. The *Get TDs* handler and the *TDs Bus1* enable direct communication between the master core and the *Task Maestro*, avoiding off-chip communication overhead, which is one of the scalability limiting factors of Nexus[108]. A similar dedicated bus, the *TDs Bus2* is shown in Figure 3.2 and is used for communication between the *Task Maestro* and worker cores.

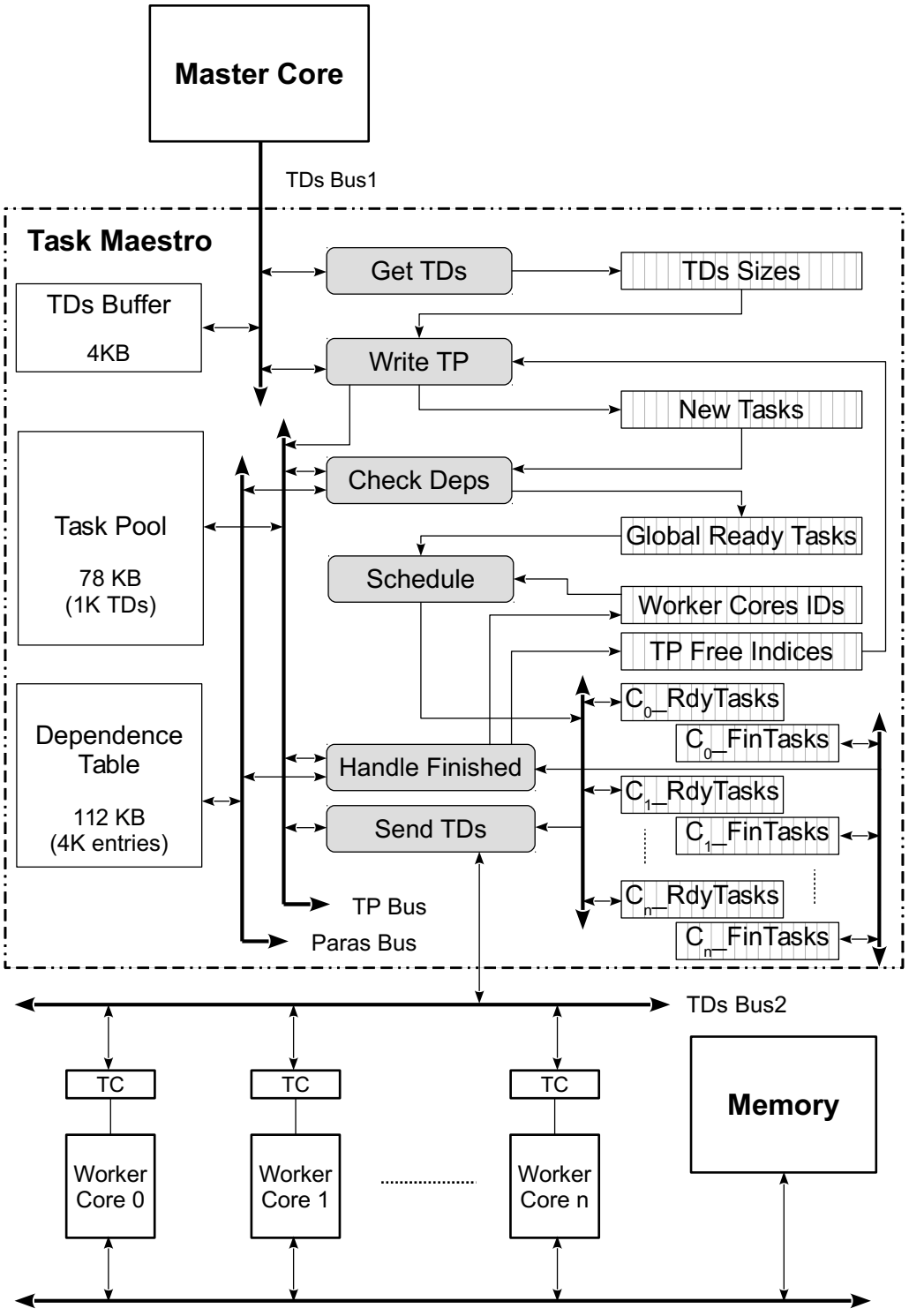


Figure 3.2: Nexus++ block diagram.

After having received a *Task Descriptor*, the *Get TDs* handler writes its size (indicating the number of parameters the task has) to a FIFO list called the *TDs Sizes* list. If this list is full, the *Get TDs* handler does not set the *acknowledge* signal in response to future incoming *request* signal from the *Master Core*, causing the latter to stall and stop sending new *Task Descriptors*.

Communication and synchronization between the different hardware handlers inside the *Task Maestro* are done using FIFO lists (such as the *TDs Sizes* list). Those FIFO lists are SystemC constructs that have useful properties. Reading or writing any of the FIFO lists will generate a list_(read/written)_event respectively. Reading is from the head of the list, and causes the read entry to be deleted from the list, whereas writing is done at the tail of the list.

Those lists pipeline the work of the different hardware handlers. A handler will stall if it tries to read/write from/to a FIFO list that is empty/full respectively.

2. **Storing tasks:**
   Once the *TDs Sizes* list is written, it triggers the *Write TP* handler, which reads the size of the recently received *Task Descriptor* from the *TDs Sizes* list, then it reads the *Task Descriptor* from the *TDs Buffer*, appends some meta data to it, and finally writes it to the *Task Pool*, the main task storage structure in Nexus++. In this prototype, a *Task Descriptor* packet received from the *Master Core* does not have special start or stop characters that mark one *Task Descriptor* from the other, this is why the sizes of the *Task Descriptors* are stored in the *TDs Sizes* list.

   The full format of the *Task Pool* is shown in Table 3.1. The 1^{st} column in Table 3.1 is the index at which tasks are stored. This index is determined by the *Write TP* handler, which reads the *TP Free indices* list, that stores initially all indices of the *Task Pool*. After the completion of a task, its *Task Pool's* index is written back to the *TP Free indices* list.

   Inside Nexus++, a task is identified by its *Task Pool* index. This is important to directly address a specific entry in the table, rather than searching the table for that entry. That is why a task's index is also stored in the *Task Descriptor* as shown in the *tp_i* column of Table 3.1.

Index	busy	tp_i	*f	DC	nD	nP	P_1	P_2	...	P_8 or ptr_next Dummy
17	0	17	0xABCD	0	0	8	1A/4/in	2A/4/in	...	1B/4/out
98	0	98	0xDCBA	1	1	10	1B/4/in	2B/4/inout	...	99/.../...
99	0	99	-	-	-	-	8B/4/in	9B/4/out	10B/4/out	-

Table 3.1: The Task Pool. (tp_i: TP index, *f: func. ptr, DC: dependence count, nD: num. dummy entries, nP: num. parameters, P_x: Parameter$_x$).

The *busy* column of a *Task Descriptor* is a boolean flag indicating whether this *Task Descriptor* is currently under processing by one of the handlers of the *Task Maestro* or not. This is to ensure exclusive access to any entry in the *Task Pool* at a certain time, and hence, prevent deadlocks. No race condition can occur on the *busy* flag itself, and this will be justified when explaining the process of handling finished tasks in Section 3.1.2.

The **f* column of a *Task Descriptor* indicates the function pointer of that task. The *DC* column stands for *Dependence Counter*, which records how many dependences must be fulfilled before this task can be scheduled to run, i.e., how many inputs of this task are outputs of older tasks.

The *nD* stores the *number of dummy entries* that are linked to this *Task Descriptor*. Adding dummy entries to the *Task Pool* is the mechanism used to overcome the limit on the number of inputs/outputs a task can have. Applications are different from one another in the number and size of tasks they can generate. In addition to differences in task granularity, tasks can have arbitrary number of parameters. When designing a system to deal with such tasks, the maximum number of parameters can be set to a large number in order to cover a wide range of tasks. This however can result in an impractical system size. This becomes more critical when designing a hardware unit to handle tasks. To provide virtually an unlimited number of parameters per task, dummy entries are reserved in the *Task Pool* for tasks that have large number of parameters. This mechanism is explained in detail in Section 3.1.3.

Columns *nP* and the following ones indicate the number of inputs/outputs, and their information, respectively. An input/output of a task is stored in the format: *(base memory address, size, and access mode)*, where the access mode can be either input, output, or inout.

3. **Resolving tasks' dependencies:**
   Once the *Write TP* handler has finished storing a task in the *Task Pool*, it writes this task's *ID* (its *Task Pool's* index) in a FIFO list called the *New Tasks* lists shown in Figure 3.2, the event that triggers the *Check Deps* handler. The latter handler is responsible for checking whether the new submitted task is ready or not, by checking the newly submitted task's inputs/outputs against all those of the previously submitted tasks. The task dependence graph is stored inside the

Dependence Table. The process of dependency resolution is described in detail in Section 3.1.2.

4. **Scheduling tasks:**
   Once a task is found ready by the *Check Deps* handler, it writes its *ID* to a FIFO list called the *Global Ready Tasks* list. This event triggers the *Schedule* handler, which is responsible for scheduling ready tasks onto the worker cores.

   Another FIFO list called the *Worker Cores IDs* list contains initially all worker cores IDs. The *Schedule* handler reads the latter FIFO for a worker core ID and schedules the last found ready task on this worker core. This simple round-robin scheduling mechanism achieves load balancing between cores, since whenever a core finishes running a task, the core's ID is written back at the tail of the *Worker Cores IDs* list.

   The *Task Maestro* has two FIFO lists for each worker core. The first one is called the $C_i RdyTasks$ (Core$_i$ Ready Tasks) list, and the second one is the $C_i FinTasks$ (Core$_i$ Finished Tasks) list. Scheduling a task on a core is done by writing the task's *ID* in that core's $C_i RdyTasks$ list. $C_i FinTasks$ lists are used later upon completion of tasks. In this prototype of Nexus++, simple distributed queues are used to communicate with the worker cores, since the focus at this point of time is on the *Task Maestro* and it's evaluation.

5. **Send ready tasks to worker cores:**
   Once the *RdyTasks* list of a certain core is written, the *list_written_event* is communicated to the corresponding worker core, via a simple unit called the local *Task Controller* (TC), which is integrated per each worker core. The *Task Controller* is mainly responsible for communication with the *Task Maestro*, and to enable buffering of tasks according to the *Task Controller* buffering depth. If the buffering depth equals m, then the *Task Controller* will buffer $m - 1$ tasks while the worker core is busy executing one task. A buffering depth of 2 implies double buffering, whereas that of 1 implies no buffering.

   A *Task Controller* contains four hardware handlers, namely the *Get TD*, *Get Inputs*, *Run Task*, and *Put Outputs* handlers, as shown in Figure 3.3. The first of them is the *Get TD* handler, which is triggered upon writing a new task ID to the corresponding core's *RdyTasks* list. The *Get TD* handler is responsible for fetching the function pointer of the ready task from the *Task Maestro*. This

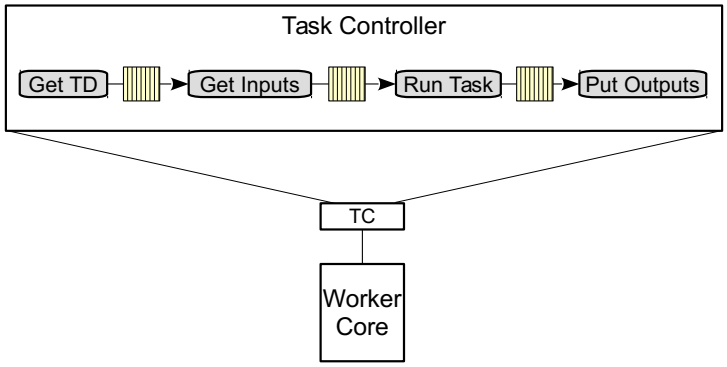


Figure 3.3: The *Task Controller* block diagram.

is done by activating a request signal to the *Task Maestro*; the event that is handled by the *Send TDs* handler in the *Task Maestro*.

The *Send TDs* handler works in a round-robin fashion. It checks all the requests from the different *Task Controllers*, and whenever it finds an active one, it reads the *RdyTasks* list corresponding to the incoming active signal and gets the ready task *ID*. Since a task *ID* is the index at which it is stored in the *Task Pool*, the *Send TDs* handler reads the *Task Descriptor* at that index directly without searching the *Task Pool* and sends it over the *TDs Bus2*, shown in Figure 3.2 to the requesting *Task Controller*.

After the *Send TDs* handler has sent the requested *Task Descriptor* to the requesting worker core, it writes the task *ID* to that core's *FinTasks* list, which is important upon task completion as will be shown later.

Sending tasks to worker cores on requests from the local *Task Controllers* ensures that the *Send TDs* handler in the *Task Maestro* will not waste any clock cycle waiting for a local *Task Controller*, due to for example a handshaking protocol or full buffer at that local *Task Controller*.

6. **Run tasks:**
   After getting a task from the *Task Maestro*, the *Get Inputs* handler at the *Task Controller* side, prefetches the task code and inputs from memory. Then, the *Run Task* handler simulates running the task by waiting for a certain time indicating by the application trace, and finally the *Put Outputs* handler writes the outputs back to memory, and notifies the *Task Maestro* of task completion. Since task

execution is simulated, the *Put Outputs* handler is necessary to simulate writing task's results.

The different hardware blocks in the *Task Controller* work in a pipelined fashion so that each block works in parallel to the block after it. For example the *Get TD* block starts getting the *Task Descriptor* of a new task while the *Get Inputs* block is getting the inputs of the previously fetched *Task Descriptor*. This happens of course given that there are some new ready tasks in the corresponding core's *RdyTasks* list. Of course, the pipeline stalls when the internal buffers of the *Task Controller* are full.

7. **Finalize tasks, and update the task graph:**
   The *task-finished* notification signals from the local *Task Controllers* are handled by the *Handle Finished* handler in the *Task Maestro*. The *Handle Finished* handler also works in a round-robin fashion; it continuously checks the notification signals from the different *Task Controllers*, and whenever it finds an active one, it performs two things: first, it acknowledges the corresponding *Task Controller* of the observation of its *task-finished* signal, so the *Task Controller* deactivates its *task-finished* signal consequently.

   The second thing the *Handle Finished* handler performs is that it reads the *FinTasks* list of the corresponding worker core. The value read is the *ID* of the finished task, since the *FinTasks* list was written by the *Send TDs* handler immediately after having sent the *Task Descriptor* to the corresponding worker core. Therefore, it is assumed that the order at which tasks are sent to a worker core, is the same order tasks would finish.

   The main goal of having the *FinTasks* lists is to minimize the communication between the *Task Maestro* and the different *Task Controllers*, and also to reduce the time of handling a finished task, since the *Task Maestro* does not need to get the finished tasks *IDs* from the *Task Controllers*.

   After reading the finished task *ID*, the *Handle Finished* handler reads the input/output list of the finished task from the *Task Pool*, updates the *Dependence Table* and kicks off pending tasks, if any. Finally, the *Handle Finished* handler deletes the task from the *Task Pool*, adds the task *ID* to the *TP Free indices* list, and adds the worker core *ID* to the *Worker Cores IDs* list.

Since the system has only one *Task Maestro* and multiple *Task Controllers*, choosing the round-robin mechanism to communicate the *Task Descriptors* to the *Task Controllers* and to handle the *task-finished* notifications ensures that the *Task Maestro* is not blocked by any of the *Task Controllers*.

Moreover, since inside Nexus++ any task is identified by the index at which it is stored in the *Task Pool*, the size and access time of the different tables and FIFO lists are reduced, since no search operation takes place. Furthermore all events and notifications are one-bit signals, which ensures low communication overhead between the *Task Maestro* handlers and the *Task Controller* handlers.

Both Nexus [109] and Nexus++ provide dependency resolution. However, Nexus can only deal with tasks with a limited number of inputs/outputs. Moreover, Nexus can deal with dependency patterns where only few, limited number of tasks depend on a certain task. In addition, Nexus proposed *Task Controllers*, but did not implement them. Nexus++ solves the above limitations as described next.

3.1.2 Dependency Resolution

Dependency resolution between tasks is accomplished inside the *Task Maestro* by the *Check Deps* handler, *Handle Finished* handler, and the *Dependence Table* along with the *Dependence Counter* associated with every *Task Descriptor* in the *Task Pool*.

Currently, dependencies between tasks are limited to comparing the base addresses of the inputs/outputs of the different tasks. For example, if a task T_1 is writing address A and T_2 is reading the same address, then, given that T_2 was submitted to the *Task Maestro* after T_1, T_2 depends on T_1 and thus, cannot be scheduled to run before T_1 finishes execution.

The *Dependence Table*

The *Dependence Table* is shown in Table 3.2. It is the place where dependence information is stored. Each input/output that is accessed by a task will have an entry in the *Dependence Table* indicating its access mode, and a *Kick-Off List* that contains the *IDs* of the tasks waiting for this address to be produced before they can run.

hAddr	v	fAddr	Size	isOut	Rdrs	ww	n_v	n_i	p_i	h_D	l_D	T_{ID}	...	T_{ID} or ptr_next Dummy
0xA	1	0x1A	4	1	0	0	0	-	-	0	-	T_2	-	-
0xB	1	0x1B	4	0	1	1	0	-	-	0	-	T_{10}	-	-
0xC	1	0x1C	4	1	0	0	1	111	-	1	333	T_{20}	...	222
0x111	1	0x2C	4	1	0	0	0	-	C	0	-	T_{50}	...	-
0x222	1	0x1C	4	-	-	-	-	-	-	1	333	T_{27}	...	333
0x333	1	0x1C	4	-	-	-	-	-	-	0	-	T_{34}	...	-

Table 3.2: The Dependence Table. (hAddr: hash address, fAddr: full address, isOut: is output, Rdrs: readers counter, n_v: next is valid, n_i: next entry index, p_i, prev. entry index, h_D: has dummy entries, l_d: last dummy entry index, ww: a writer waits, T_x: $Task_x$).

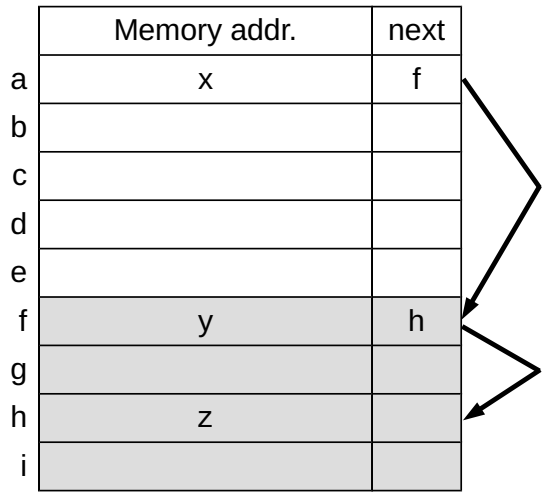


Figure 3.4: A variable-length linked-list structure formed inside the *Dependence Table* upon inserting two memory addresses x and y that map to the same location in the *Dependence Table* described in Chapter 3.

The *Dependence Table* is a hash table with a simple separate chaining hash collisions resolution algorithm. The hash function used in this prototype is a simple one shown in Equation 3.1.

$$h = addr \ \% \ [Dependence \ Table \ size] \tag{3.1}$$

Basically, the addresses in a task's input/output list are truncated to the least significant bits equal to the $\log_2(Dependence\ Table\ size)$. If a collision occurred between memory locations x and y on a certain *Dependence Table* entry, a for example, memory location (y) will be assigned another entry in the *Dependence Table*, b for example, and a link to b will be inserted in a, creating a linked-list structure inside the *Dependence Table*, as shown in Figure 3.4.

Although this hash function will not lead to a fairly even distribution in real application executions, this is not the case in this SystemC prototype, where the task's input/output addresses are generated synthetically by the test bench. However, the hash function is configurable and can be changed when necessary. In case of a hash collision, the *Dependence Table* has an overflow section where the addresses causing a collision can be inserted and connected in a linked-list style.

The different fields of the *Dependence Table* are shown in Table 3.2. The first column *hAddr* is the hash address, followed by a valid bit in the *v* column, followed by the full memory address in the *fAddr* column. Size and access mode of this memory address are stored in the *Size* and *isOut* columns respectively.

The *Rdrs* column indicates the number of tasks reading-only this memory address at a certain time. The *ww* flag (stands for *a writer waits*) indicates whether a task is waiting for previous readers to finish before it can run and write this memory address. The latter case is known as the write-after-read hazard *WAR*. Although the *WAR* hazards and the write-after-write *WAW* hazards are not real dependencies and are normally resolved using renaming techniques, Nexus++ treats them as true dependencies. However, implementing address renaming techniques in Nexus++ is considered in the future work.

The *n_v, n_i*, and *p_i* columns stand for *next is valid* flag, *next index*, and *previous index* respectively, which builds up the linked-list structure inside the *Dependence Table* for entries that map to the same hash address. The *h_D* and *l_D* are the *has dummy* flag and *last dummy index* to implement the dummy entries mechanism explained in Section 3.1.3 in the *Dependence Table*, in order to overcome the limit on the number of tasks that can depend on a certain memory address. The number of dependent tasks can vary arbitrary and in order to solve this problem, multiple entries in the *Dependence Table* are reserved for one memory address, in order to distribute the tasks' IDs of the dependent tasks over the different *Kick-Off Lists* of those entries.

A *Kick-Off List* is composed of the columns $T_1 \ldots T_8$ of Table 3.2.

Resolving new tasks dependencies

Every newly submitted task to the *Task Maestro* is handled by the *Check Deps* handler, whose pseudocode is shown in Listing 3.1. Listing 3.1 shows that for each entry A in the input/output list of the *newTask*, the *Dependence Table* is looked up, and an entry for A will be reserved if A was not found. If A was found on the other hand, then an older task is already accessing A. In this case, the access modes are checked; if both the old and new tasks access A as read-only, then the new task is granted access to A. However, if the older task is writing A, then the new task T_2 is added to the *Kick-Off List* of A as shown in Table 3.2 regardless of its access mode to A , and its *Dependence Counter* is incremented.

```
1 foreach A in parameters[newTask]
2 {
3  if(A not in DT){                          //1
4   Add A to DT;
5   if(newTask read-only A){                 //2
6    DT[A].Rdrs=1;
7    DT[A].isOut = false;
8   }
9   else                                     //2'
10   DT[A].isOut = true;
11 }
12 else                                      //1'
13  if(newTask read-only A)                  //3
14   if(!DT[A].isOut && !DT[WriterWaits])    //4
15     DT[A].Rdrs++;
16   else{                                   //4'
17    DT[A].writeKickOffList(newTask);
18    TP[newTask].DC++;
19   }
20  else{                                    //3'
21   DT[A].writeKickOffList(newTask);
22   TP[newTask].DC++;
23   if(!DT[A].isOut)
24    DT[A].WriterWaits = true;
25  }
26 }
27 if(TP[newTask].DC == 0)
28  GlobalReadyTasksList.write(newTask);
```

Listing 3.1: Pseudocode of checking dependencies for the new tasks.

Finally, the *WAR* hazards are handled using the *ww* (a writer waits) flag in Table 3.2. If a task T_1 is reading B, and T_{10} wants to write B, then T_{10} is added to the *Kick-Off List* of B as shown in Table 3.2, its *Dependence Counter* is incremented, and the *ww* flag is set. Any other task that wishes to access B afterwards, regardless its access mode, will be added to the *Kick-Off List* of B, and its *Dependence Counter* is incremented.

After checking all inputs/outputs of a new task, the *Check Deps* handler checks the new task's *Dependence Counter*, if it was 0, then the task does not depend on any other older tasks, and can be scheduled to run.

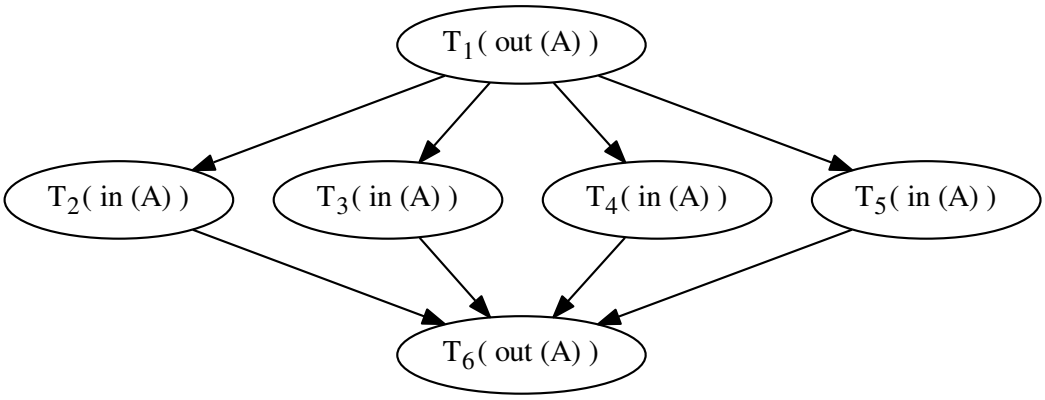


Figure 3.5: The task graph of a program in which 6 tasks access memory address A.

Handling finished tasks

Upon task completion, the *Handle Finished* handler takes action. The pseudocode of Listing 3.2 shows how the *Handle Finished* handler updates the *Dependence Table* and passes ready tasks to the *Schedule* handler.

For example, the pseudocode shows that for each entry A in the input/output list of the finished task T_1, if A is read-only, then the *Rdrs* count of A is decremented. If it becomes 0 and no writer task is waiting (*ww* flag is false), then A is deleted from the *Dependence Table*. But if the *ww* flag is true, then a pending task T_2 must exist and is read from *Kick-Off List* of A.

On the other hand, if T_1 is a writer of A, and no tasks are waiting for A, then A is deleted from the *Dependence Table*. But if there are some tasks waiting for A, then the *Handle Finished* handler will continuously read these tasks' *IDs* one after the other as long as they read-only A, until it reads a task that writes A, or the *Kick-Off List* of A is empty. Each time a reader is read from the *Kick-Off List*, the *Rdrs* count of A is incremented.

Figure 3.5 shows the resulting task graph of an example, where 6 tasks access A. T_1 writes A, T_2 to T_5 read A, whereas T_6 writes A. The tasks are submitted starting with T_1 and ending with T_6 according to their numbers.

In the example of Figure 3.5 and while T_1 is running, the *isOut* field of A in the *Dependence Table* is set. T_2 to T_6 are added to the *Kick-Off List* of A. When T_1

```
1 foreach A in parameters[FinishedTask]
2 {
3  if(FinishedTask read-only A){          //1
4   DT[A].Rdrs--;
5   if(DT[A].Rdrs==0)
6    if(DT[A].WriterWaits){               //2
7     wTask=DT[A].readKickOffList();
8     DT[A].isOut=true;
9     DT[A].WriterWaits=false;
10    TP[wTask].DC--;
11    if(TP[wTask].DC==0)
12     GlobalReadyTasksList.write(wTask);
13   }
14   else                                 //2'
15    delete DT[A];
16 }
17 else{                                  //1'
18  bool RdrsOnly = true;
19  while(RdrsOnly){
20   if(DT[A].Rdrs==0 &&
21      DT[A].KickOffList empty){         //3
22        delete DT[A];
23        RdrsOnly = false;
24   }
25   else{                                //3'
26    wTask=DT[A].readKickOffList();
27    TP[wTask].DC--;
28    if(wTask reads-only A){             //4
29     DT[A].isOut = false;
30     DT[A].Rdrs++;
31    }
32    elsif(DT[A].Rdrs==0){               //4'
33     DT[A].isOut = true;
34     RdrsOnly = false;
35    }
36    else{                               //4"
37     DT[A].WriterWaits=true;
38     RdrsOnly = false;
39     DT[A].writeHead_KickOffList(wTask);
40    }
41    if(TP[wTask].DC==0)
42     GlobalReadyTasksList.write(wTask);
43   }
44  }
45 }
46 }
```

Listing 3.2: Handling finished tasks pseudocode.

finishes execution, the *Handle Finished* handler reads the T_2 to T_5 tasks in *A's Kick-Off List* since they all read-only A, resets the *isOut* flag, increments the *Rdrs* count to 4, and stops reading tasks from the *Kick-Off List* when reaching T_6 since it writes A. In this case, the *Handle Finished* handler sets the *ww* flag of A in the *Dependence Table*. When tasks T_2 to T_5 finish execution, the *Rdrs* count is reset to 0, T_6 is fetched from the *Kick-Off List* of A and the *isOut* field of A in the *Dependence Table* is set. Finally, when T_6 finishes execution, the *Handle Finished* handler deletes A from the *Dependence Table* if its *Kick-Off List* is empty.

Of course, every time a task is read from a *Kick-Off List*, its *Dependence Counter* is decremented, and if it becomes 0, the task *ID* will be written to the *Global Ready Tasks* list to be scheduled to run.

Avoiding race conditions

In the *Task Pool*, the *busy* flag is used to ensure that the different handlers access a certain *Task Descriptor* exclusively. More precisely, to prevent race conditions that might occur when accessing the *Dependence Counter* of a certain task.

This might happen as in the following scenario: if T_1 has 20 parameters, the first of them is A which is an output of a previous task T_2. The *Check Deps* handler will add T_1 to the *Kick-Off List* of A. If T_2 is finished before the *Check Deps* handler has finished processing all the 20 parameters of T_1, then the *Handle Finished* handler will try to access the *Task Pool TP[*T_1*]* in order to decrement its *Dependence Counter* whilst the *Check Deps* might also be trying to increment it. This potential race condition is prevented by the *TP[*T_1*].busy* flag, since the *Handle Finished* handler will not access a *Task Descriptor* in the *Task Pool* if it was busy flag was set.

As shown in the previous scenario, the *Check Deps* and *Handle Finished* handlers can never be active at the same time for the same *Task Descriptor*. The *Check Deps* always starts first, and the *Handle Finished* will start later. Since the *Handle Finished* handler checks the *busy* flag before trying to access any *Task Descriptor*, this ensures that the *Handle Finished* will never access a *Task Descriptor*, before the *Check Deps* has finished processing that *Task Descriptor*.

Dependency resolution in Nexus++ is more efficient than that in Nexus [109], since Nexus++ uses fewer and simpler tables and *Kick-Off Lists*. Nexus++ has only one

table (*Dependence Table*) to maintain the task graph, and using the *Task Pool's* indices as task *IDs* eliminates the need to search the tables. In Nexus, on the other hand, three tables (containing two *Kick-Off Lists*) are used and are accessed always for all kinds of scenarios. For example, if T_1, T_2, T_3 are a writer, a reader, and a writer of A respectively[1], then in Nexus, A will have an entry in two tables, with two *Kick-Off Lists*, and task T_3 will be added to the two *Kick-Off Lists* of A. Whereas no data is replicated in Nexus++. Hence, Nexus++ is simpler and performs fewer computations to resolve dependencies.

3.1.3 Dummy Tasks and Entries

Dummy tasks in the *Task Pool*

In a *Task Descriptor*, a task has a limited number of inputs/outputs, so applications with tasks that have more inputs/outputs cannot be executed directly on a system with Nexus. In addition, not all tasks necessarily have a number of inputs/outputs equal to the *Task Descriptor's* limit, which yields a poor memory utilization.

Nexus++ solves this problem by introducing dummy tasks. A dummy task will not be executed, it just takes the form of a task by having an entry in the *Task Pool*, only to store the inputs/outputs that did not fit in the original input/output list.

Figure 3.6 shows a scenario to demonstrate the need for dummy tasks. If T_x has $2n$ outputs, and a *Task Descriptor* can only store n of them, then dummy tasks (D_1 and D_2) are created having their inputs/outputs as those that did not fit in the *Task Descriptor* of T_x. A dummy task is simply a pointer that replaces the last entry of an input/output list.

In Table 3.1, this mechanism is accomplished using the *nD* (number dummy) column along with the last column (P_8 *or ptr_next Dummy*) of a *Task Descriptor*. Assuming that a *Task Descriptor* allows up to n inputs/outputs per task, if a task has more than n inputs/outputs, then this task is distributed to occupy multiple *Task Descriptors* in the *Task Pool*.

The number of the extra *Task Descriptors* needed is stored in the *nDummies* column of the original entry, as shown in the example in Table 3.1. The *Task Descriptor* at

[1]The detailed example is shown in Figure 6.20 in [108]

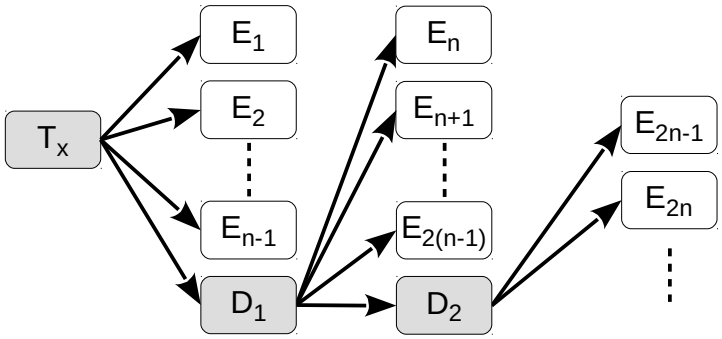


Figure 3.6: Dummy Tasks/Entries added to the *Task Pool*/*Dependence Table*.

index 98 has 10 inputs/outputs, which is more than the limit of 8 (for this example) per *Task Descriptor*. That is why a new entry is occupied by this task, namely the *Task Descriptor* at index 99. The entry at index 98 has 1 in its *nDummies* field, indicating that this task occupies in total 2 *Task Descriptors*, and the last entry in its input/output list now points to index 99. The *Task Descriptor* at index 99 is called a *Dummy Task*, since it is not a true task, rather a placeholder to store the extra input/output information that did not fit in the original entry. Reserving dummy tasks for a certain task is done by the *Write TP* handler.

Using dummy tasks, Nexus++ can store tasks of arbitrary number of inputs/outputs, by implementing a linked-list structure inside the *Task Pool*. Although this solves the problem of having a fixed, limited number of inputs/outputs per task, the maximum number of inputs/outputs is still bounded by the size of the *Task Pool*.

Dummy entries in the *Dependence Table*

The same principle can be deployed in the *Dependence Table* shown in Table 3.2, where the *Kick-Off List* has a limited size, thus restricting the number of tasks that might depend on a certain memory address. As a solution, Nexus++ adds dummy entries to the *Dependence Table* to extend the *Kick-Off List* of a certain entry.

Figure 3.6 can be interpreted as follows: assuming that $2n$ tasks depend on a single output O of T_x and given that the size of the *Kick-Off List* of the *Dependence Table* entry DT[(O)] is n, then the $2n$ tasks cannot be fit in the *Kick-Off List*.

A stall, stopping the process of inserting further tasks to the system, is not an optimal solution because the *Dependence Table* might still have empty slots. Furthermore, other tasks cannot be added to the system since they might depend on an output of one of the extra tasks that did not fit in the *Kick-Off List* of O.

Another non-optimal solution is to increase the size of the *Kick-Off List*, since it also limits, although larger, the *Kick-Off List* size. It also wastes memory since not all outputs will be inputs for a number of tasks that is equal to the *Kick-Off List* size.

Furthermore, implementing the worst-case scenario is not scalable. The proposed solution here is, therefore, to insert some dummy entries in the *Dependence Table*, namely D_1 and D_2, resulting in a chain of *Kick-Off Lists*. Similar to dummy tasks in the *Task Pool*, the number of tasks that can depend on a certain memory address is bounded by the size of the *Dependence Table*, though not fixed at a certain value for all *Dependence Table's* entries.

In Table 3.2 on page 64, an example is shown. Memory address *0x1C* is currently being written by a certain task T_1, and the number of tasks that are waiting in the *Kick-Off List* of *0x1C* does not fit in a single *Kick-Off List*. That is why the *h_D* flag of *0x1C* is set, and the last entry in the *Kick-Off List* of *0x1C* points to address 222, which contains also some tasks *IDs*, and also has another dummy entry at address 333 of the *Dependence Table*, where the rest of the waiting tasks are registered. The parent entry of memory address *0x1C* stores always the address of the last dummy entry of the chain. This is important when adding new tasks to the *Kick-Off List* of *0x1C*, since they will be added to the last dummy entry, without the need to traverse the whole chain until the last one is reached.

In Figure 3.6, having only one dummy entry per *Kick-Off List* is efficient, since when reading a *Kick-Off List*, the *Task Maestro* might read only one entry (in case of a writer task), or more. Reading only one entry from the *Kick-Off List* chain is simply reading the first entry of the first *Kick-Off List* in the chain. On the other hand, allowing more than one dummy entry per *Kick-Off List* (forming a linked-list structure of *Kick-Off Lists*) will result in multiple lookups in order to reach the leaf *Kick-Off List*, which is an extra overhead.

Reading tasks *IDs* from the *Kick-Off List* of a certain memory address happens from the head of the first *Kick-Off List* of the chain. Whenever all tasks are read from the first *Kick-Off List*, this entry's data (except the *Kick-Off List* and the *h_D*

fields) will be copied to the next dummy entry so that it becomes the new head. For example, memory address *0x1C* occupies 3 entries (at DT[0xC, 0x222, and 0x333]) in Table 3.2 on page 64. When all items in the *Kick-Off List* of DT[0xC] are read, this entry will be invalidated, and the head entry of *0x1C* will reside at DT[0x222]. This way, the *Dependence Table* is efficiently utilized, since DT[0xC] can now be reused by other memory addresses, even before memory address *0x1C* is totally removed from the *Dependence Table*. This also allows direct (and hence, fast) access to the first *Kick-Off List*, since it always resides at the head entry of a memory address.

Dummy tasks are injected by the *Task Maestro* when needed at runtime. They utilize memory well, and are scalable. The compiler could also add dummy tasks when it discovers that a task has more inputs/outputs than the maximum. However, the master core then would have to generate and submit more *Task Descriptors*, and Meenderinck [108] in his work indicates that eventually the master core forms the bottleneck. Furthermore, the compiler cannot add dummy entries to the *Dependence Table* since it depends on runtime information which is not available to the compiler. For these reasons it is the *Task Maestro* who is responsible for adding the dummy tasks and entries.

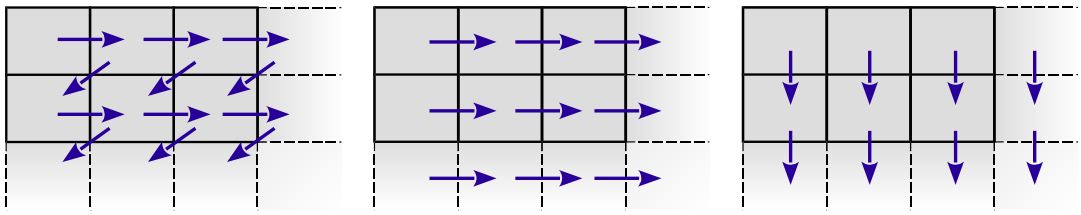


Figure 3.7: Dependency patterns (120 × 68 blocks): (a) ramp effect, (b, c) fixed # of parallel tasks.

3.2 Experimental Setup

3.2.1 Benchmarks

Several benchmarks are used to evaluate Nexus++. First, a trace of a parallel H.264 decoder decoding one full HD frame on a Cell Broadband Engine processor [125] is used, consisting of 8160 tasks in total. The trace consists of tasks input/output information, tasks execution times and the time they have spent reading/writing their inputs/outputs from/to memory. On average a task spends 7.5μs for accessing the off-chip memory and 11.8μs for execution [42]. The benchmark processes a matrix of 120 × 68 macroblocks and the dependency pattern is shown in Figure 3.7(a) [156]. Tasks are generated in the serial execution order, which is from left to right and from top to bottom. Initially there is only one task ready for execution, but this number increases until halfway execution, after which it decreases again [111]. This ramping effect influences the average amount of parallelism available in the benchmark and therefore its scalability too.

To evaluate Nexus++ for a range of dependency patterns, two additional synthetic benchmarks were created. Their dependency patterns are shown in Figure 3.7(b) and (c). An additional benchmark without dependencies, i.e., has all-independent tasks, was also created in order to measure the maximum scalability of Nexus++. All the synthetic benchmarks feature 4 parameters per task.

In contrast to dependency pattern (a), the dependency patterns depicted in Figure 3.7(b) and (c) do not suffer from the ramping effect. Instead, these dependency patterns provide a constant number of parallel tasks. In (b), however, the dependency pattern has the same direction as the order in which tasks are generated. As a con-

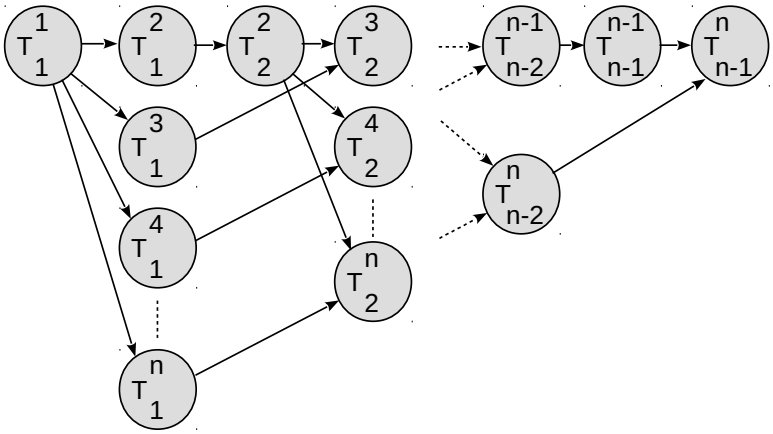


Figure 3.8: Dependency pattern for the Gaussian elimination benchmark. T_i^j: i, j row and column numbers respectively.

sequence, the amount of effective available parallelism can be reduced by the speed of tasks' insertion into the *Task Pool* as well as the size of the *Task Pool*, since when the table is full, tasks of the first row have to be executed to make room for the other tasks in the next rows.

To validate the dummy tasks/entries approach, the task graph of Gaussian elimination with partial pivoting [161] was used. In this benchmark, the number of tasks that depend on certain outputs depends on the size of the input matrix as depicted in the dependency pattern of Figure 3.8, assuming an $n \times n$ matrix.

The execution starts with one task (T_1^1), on which $n-1$ tasks ($T_1^2 ... T_1^n$) depend. After that only a single task (T_2^2) can execute, and then $n-2$ tasks, etc. The total number of tasks is proportional to the matrix size, and equals $\frac{n^2+n-2}{2}$, where n is the matrix dimension. Each task performs a number of FLOPs. This number represents the weight W of a task, which is equal to [161]:

$$W(T_i^j) = \begin{cases} n+1-i & FLOPs \quad \text{if } i = j \\ n-i & FLOPs \quad \text{if } i < j \end{cases} \tag{3.2}$$

where i, j are the row and column numbers respectively. Hence the duration of a task T_j^i equals $W(T_i^j)$, divided by the *GFLOPS* of one core. Each task also reads $W(T_i^j)$ *floating point* numbers from memory, and writes the same number back when finished.

Matrix dimension	# Tasks	Average task weight (FLOPs)
250	31374	167
500	125249	334
1000	500499	667
3000	4501499	2012
5000	12502499	3523

Table 3.3: Gaussian elimination tasks for different matrix sizes.

Some tasks in the Gaussian elimination benchmark are very small (a few FLOPs), but as shown in Equation (3.2) and Figure 3.8, the number of tasks of a certain weight is directly proportional to the weight itself. So the majority of tasks are relatively coarse-grain, and only a small number of tasks are fine-grain. For example, performing Gaussian elimination on a matrix size of 1000×1000 generates (as shown in Figure 3.8) one task of weight 1000, then 999 tasks each of weight 999, then a single task of weight 999, and 998 tasks each of weight 998, and so on until at the end, only a single task of weight 1 is generated. Table 3.3 gives an overview about the number and granularity of the Gaussian tasks for different matrix sizes.

3.2.2 Simulation Environment

Nexus++ is simulated using the *Task Machine*, a SystemC simulator of a task-based, trace-driven multicore system. The *Task Machine* is a fully configurable system that is designed to match modern real systems. Among the configurable parameters are the number of cores, the core clock frequency, and on chip and off chip memory access times. There are neither real cores in the *Task Machine* nor real task execution.

Tasks' information are read from experimental traces, which include the input/output information and also tasks' execution and memory access times. Thus task execution is simply simulated by waiting for a certain time equivalent to the execution time read from the experimental trace. The list of parameters and their values are shown in Table 3.4.

Nexus++ is simulated assuming a clock cycle time of 2 *ns*, which equals a clock frequency of 500 *MHz*. The *Task Maestro* tables and the FIFO lists are on-chip storage and therefore their access times are relatively fast. The hash table access time

System Parameter	Value
Cores clock freq.	2.0 GHz
Nexus++ clock freq.	500 MHz
On Chip Access Time	2 *ns*
Off Chip Access Time	12 *ns*
On chip bus bandwidth	2 GB/s
Memory bandwidth	10.67 GB/s
Task Descriptor (TD) size	78 Byte
Task Pool size	78 KB (1K TDs)
No. Parameters per *TD*	8
Dependence Table entry size	28 Byte
Dependence Table size	112 KB (4K entries)
Kick-Off list size	8 task IDs
TDs Sizes list size	1 KB
New Tasks list size	2 KB
TP Free Indices list size	2 KB
Global Ready Tasks list size	2 KB
Worker Cores IDs list size	2 KB
C_x RdyTasks list size	4 Bytes
C_x FinTasks list size	4 Bytes

Table 3.4: System parameters.

equals the on-chip access time multiplied by the number of lookups required per 1 access.

The traces recording execution and communication times per task are generated after fine-grain (1 macroblock per task) parallel H.264 decoding on a Cell processor [125]. Thus, the experiments are assuming a local-stores, shared-memory architecture. Nevertheless, Nexus++ concept can be applied to any other multicore architecture. For the synthetic benchmarks shown in Figure 3.7 and the benchmark with all-independent tasks, task durations are also from the H.264 decoding trace.

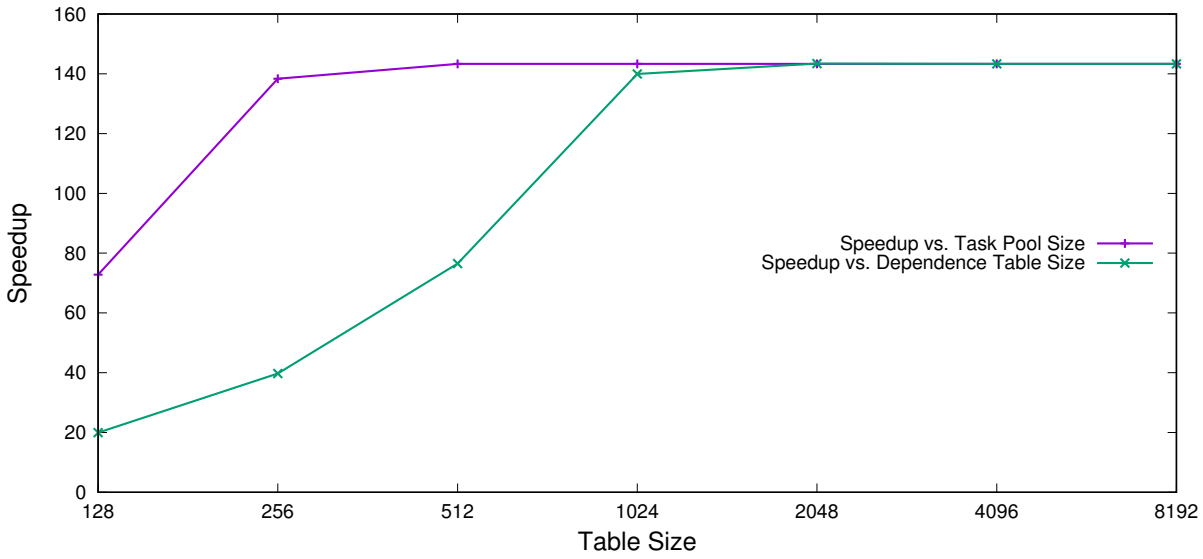


Figure 3.9: The speedup achieved when varying the size of the *Task Pool* and fixing the size of the *Dependence Table* and vice versa.

3.2.3 Design Space Exploration

Design space exploration is performed by running the all-independent tasks benchmark on a 256-core system with double buffering, and contention-free memory. The speedup in the following experiments is calculated by dividing the serial execution time (the sum of all the tasks' duration in the trace) by the time consumed by the simulated 256-core system.

First, in order to determine the optimal *Dependence Table* size, all the other structures are configured to be very large, the *Task Pool*, for example, is configured to hold 8K *Task Descriptors* at once (given that the total number of tasks is 8160).

The first experiment in Figure 3.9 shows the speedup achieved when increasing the *Dependence Table* size, and fixing the *Task Pool* size at 8K entries. Maximum speedup equals 143× when setting the *Dependence Table* size to 2K entries or more.

The second experiment in Figure 3.9 shows the speedup when varying the *Task Pool* size, and fixing the *Dependence Table* size at 8K entries. A *Task Pool* size of 512 entries is enough to achieve a speedup of 143×, however, a 1K entries *Task Pool* is chosen to allow a larger task window.

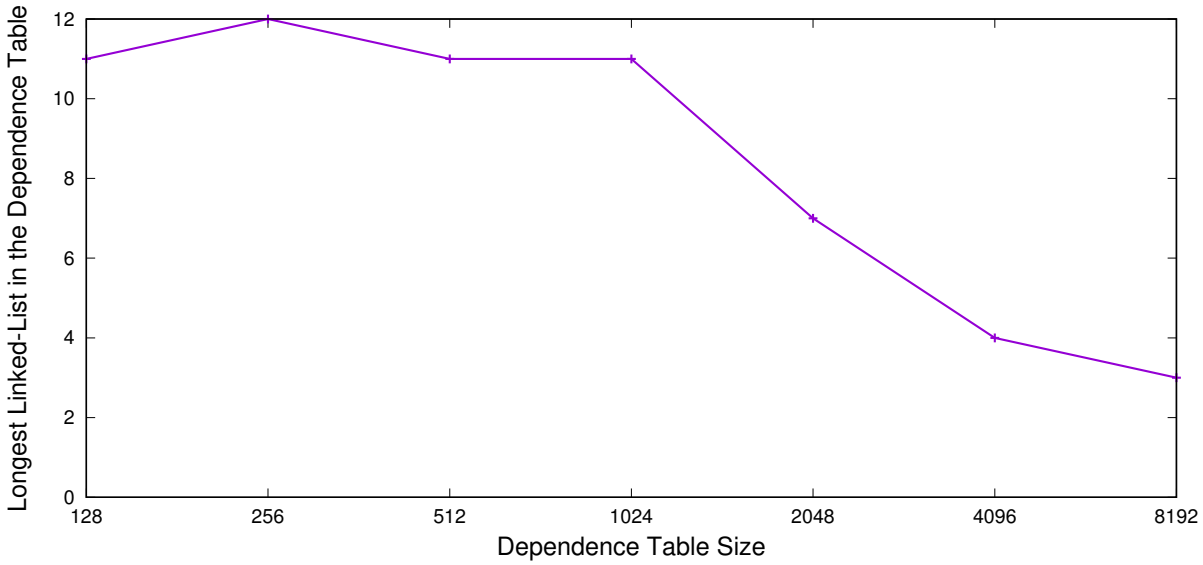


Figure 3.10: The effect of varying the *Dependence Table* size on the maximum length of the linked-list structures that are formed whenever a hash-collision occurs in the *Dependence Table*, when running the all-independent tasks benchmark.

Interestingly, the effect of varying the size of the *Dependence Table* on the speedup is more visible in Figure 3.9, when comparing the two curves in the range of 128 to 1024 entries. The synthetic benchmark includes fine-grain tasks and each task has 4 parameters. This indicates that a hash collision occurs after sending 32 tasks (having 32*4=128 parameters) in the *Dependence Table*, and thereafter maintaining the *Dependence Table* becomes more time-consuming.

This is shown in Figure 3.10 which depicts the maximum observed length of the linked-list structures resulting from inserting dummy entries in the *Dependence Table* whenever a hash collision occurs.

From the experiments shown in Figures 3.9 and 3.10, the *Dependence Table* is set to 4K entries since this size enhances shorter linked-list structures (almost half of that when the *Dependence Table* size is set to 2K entries), as longer linked-lists imply longer search time and impact the system performance significantly as shown in Figure 3.9.

Assuming 8 parameters and a total 78 Bytes per task descriptor yields a *Task Pool* size of 78 KB. The *Dependence Table*, on the other hand, should be able to hold 4K entries, as shown in Figure 3.9. Each entry size equals 28 bytes, which yields a table size of 112 KB.

Having 1K tasks in the *Task Pool*, 10 bits are needed index it and to identify a single task. This number is rounded up to multiples of a byte (i.e., 2 bytes), yields that 2KB are needed to store the IDs of 1K tasks, which is the selected size for the *New Tasks list*, the *TP Free Indices list*, and the *Global Ready Tasks list*. 1 byte is allocated to store the size of one *Task Descriptor* upon its reception from the *Master Core*. This gives a total size of 1KB for the *New Tasks list* to store the sizes of 1K *Task Descriptors*. Although setting the size of a *Task Descriptor* to 1 byte limits the number of inputs/outputs per task, but this number is configurable and is set to 1 byte in this prototype since all the benchmarks used for evaluation have small number of inputs/outputs per task.

Simulating up to 512 worker cores, requires 9 bits to assign an individual ID to each core. Rounding this number up to multiples of bytes gives a 2KB *Worker Cores IDs list* size. Assuming double buffering, a worker core should be able to store two task IDs in its *RdyTasks* and *FinTasks* lists, which yields a size of 4 bytes per list.

The sizes of the *Task Maestro* tables and lists were empirically determined. They are summarized in Table 3.4 on page 78.

3.2.4 Memory Access Latencies

The access time for the ∼100 KB on-chip memory structures (those are mainly the *Task Pool* and the *Dependence Table*) was determined using Cacti 5.3 [93, 153], and was found to be 2 *ns* for each of them. Off-chip memory (RAM) access time was also determined using the same tool, and was found to be 12 *ns* per 128 bytes RAM chunk, assuming 32-bank 1GB of RAM, which is equivalent to a maximum memory bandwidth of 10.67 GB/s. The off-chip memory is assumed to have 32 banks, each having one read/write port. Therefore, no more than 32 tasks can access the memory at a given time, and this is how contention accessing off-chip memory is modeled.

The latencies of the preparation and submission of *Task Descriptors* by the master core were estimated. These times were measured in Nexus [108] in detail. As Nexus++

avoids off-chip communication in this part, we had to compensate for this. As a result, the task preparation was set to 30 *ns*, while the task submission is not fixed since it depends on the size of the input/output list of a task.

The modeled on-chip bus is a very basic one. It is an 8-byte bus, and its bandwidth is assumed to be 2GB/s which is a typical bandwidth of the state-of-the-art on-chip buses [130].

Every time the *Master Core* wishes to submit a task to the *Task Maestro*, it arranges the task's information into 8-byte words. The first word specifies the task's ID and function pointer, and every other word specifies a single parameter (including its address, size, and access mode). The *Master Core* also sends initially a handshaking word specifying the new task's number of words, and hence, number of its parameters. In the evaluation, it is assumed that for each task submission, an initial (handshaking) bus delay of 5 cycles is needed, and that each word requires 2 cycles (2GB/s bus bandwidth) to reach the *Task Maestro*. For example, a task with 4 parameters consumes 10 cycles (20 *ns*), whereas an 8-parameter task consumes 14 cycles (28 *ns*) submission delay.

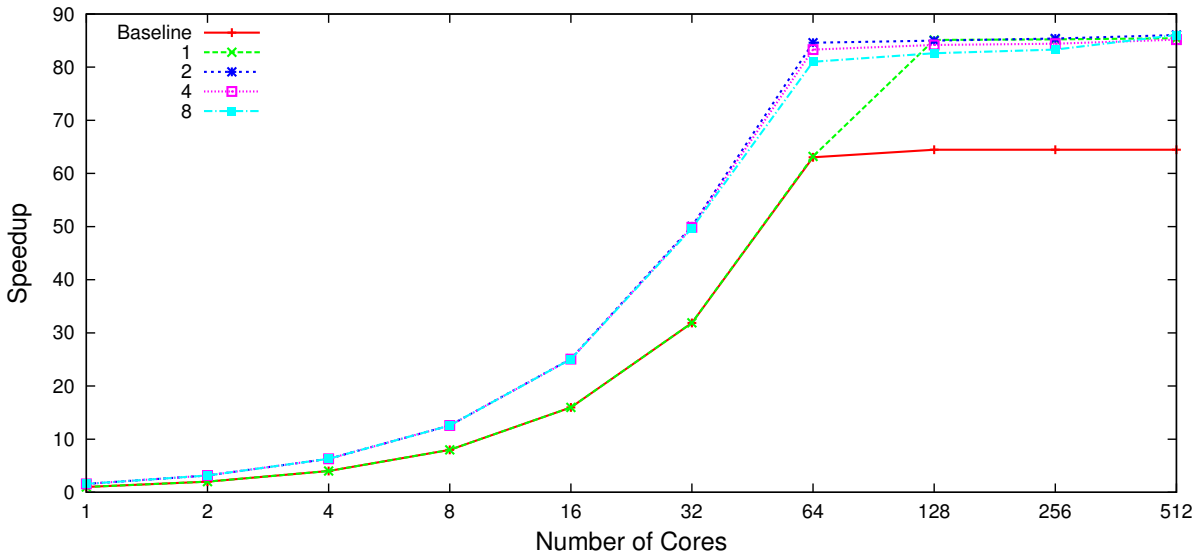


Figure 3.11: The speedup achieved by different number of cores with different buffering depths, when running independent tasks.

3.3 Evaluation Results

Nexus++ was tested under different conditions, varying the number of worker cores, the buffering depth, and with different dependency patterns.

Figure 3.11 depicts the speedup for the all-independent tasks benchmark on different number of cores and buffering depths. The baseline system is based on Nexus [109], which does not have any of the enhancements of Nexus++, i.e., no buffering of *Task Descriptors* between the *Master Core* and the *Task Maestro*, no buffering of tasks at the *Task Controller* side, and no dummy tasks or entries. In Figure 3.11, the speedup is measured relatively to the 1-core experiment of the baseline system.

Figure 3.11 shows that a speedup of 85× is achieved with 64 cores when adding double buffering ($BD = 2$). This speedup gain is larger than the number of cores because the speedup is not measured against the 1-core experiment of the same configuration (with Nexus++ enhancements enabled), but to the baseline configuration.

This baseline results differ (almost 16× speedup for 16 cores) from those shown in the baseline [108], because the average task computation and communication times are different. In our system it is assumed that these values to be 11.81 μsec and 7.5 μsec, respectively based on [42], while in [108] they are assumed to be 19 μsec and 2

μsec, respectively. Furthermore, in [108] (page 217), on-chip access time is less than one third of that in Nexus++.

Compared to other speedup curves in Figure 3.11, the speedup gain does not increase when increasing the buffer depth. On the contrary, it decreases slightly as is most visible in the 64-core experiment. This is reasonable since the *Task Maestro* stalls when it fills all *RdyTasks* lists of the worker cores. Each of these lists has a size of $BD * 2Bytes$ (the 2 Bytes in order to fit the largest task *ID*). Increasing the buffering depth results in increasing the number of tasks the *Task Maestro* has to process before it stalls, and thus, larger number of inputs/outputs will be inserted to the *Dependence Table*.

Since the *Dependence Table* is a hash table, the more entries it has, the more collisions will occur when searching for an entry, and hence, increasing the search time. The experiments show that the maximum number of collisions was 4, 7, 11, and 19 having the buffering depth equals 1, 2, 4, and 8 respectively. This is why the speedup gain decreased when increasing the buffering depth for 64 cores and more.

It can therefore be concluded that the experiment with double buffering ($BD = 2$) is sufficient and most efficient. Having this result, the speedup will be measured from now on against the single core experiment with double buffering. Accordingly, for the independent tasks benchmark, the achieved speedup equals 54× on 64 cores.

In Figure 3.11, the only difference between the baseline experiment and the experiment with $BD = 1$ is that there is no *Task Descriptor* buffering between the *Master Core* and the *Task Maestro* in the baseline experiment. The effect of this can be seen when the number of worker cores is larger than 128, where - in the baseline experiment - the *Master Core* is not able to submit enough tasks for the *Task Maestro* to keep all worker cores busy. This demonstrates how buffering of the *Task Descriptors* improves the scalability to larger numbers of cores.

The independent tasks benchmark was also performed on different multicore systems and different buffering depths, but this time assuming a perfect memory system, i.e., no memory contention can occur. The observed speedup was 143×, and the benchmark scaled up to 256 cores rather than to 64 cores (with 54× speedup) in the experiment with memory contention modeled.

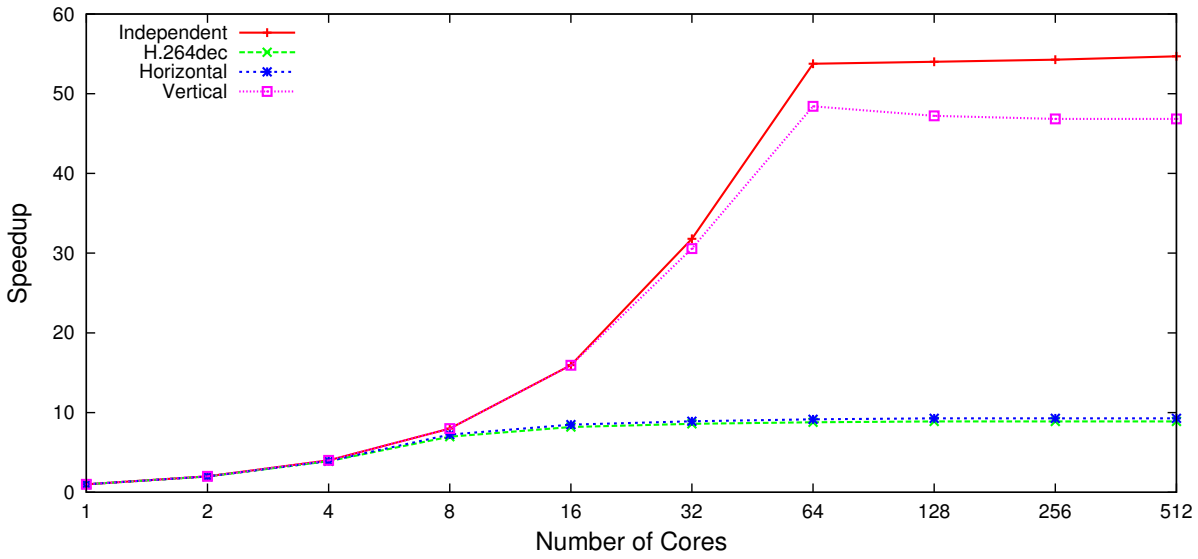


Figure 3.12: The speedup achieved by different number of cores running tasks with dependencies shown in Figure 3.7.

The baseline behavior was almost the same as that with memory contention modeled, due to the lack of task double buffering. To calculate the maximum achievable speedup, in addition to assuming memory contention free, task preparation delay is disabled, and the resulting speedup was 221× using 256 cores.

Although 256/512 cores are large numbers, these experiments are used to measure the maximum scalability of Nexus++ using the independent tasks benchmark. Furthermore, the Gaussian elimination benchmark can run on such large number of cores, when applied to large matrices.

Figure 3.12 shows the achieved speedup for the benchmarks illustrated in Figure 3.7 on page 75. As before, 8160 tasks are simulated with execution and communication times obtained from a parallel H.264 decoder [42]. Furthermore, since the previous experiments has shown that double buffering is most efficient, the next experiments assume a buffering depth of 2. The speedup is measured against the single core experiment of Nexus++ (double buffering enabled).

The effect of the order of spawning tasks on the speedup is shown by noticing the speedup difference between the benchmarks with horizontal and vertical dependencies illustrated in Figures 3.7(b) and 3.7(c) on page 75. Although the *Task Pool* is larger

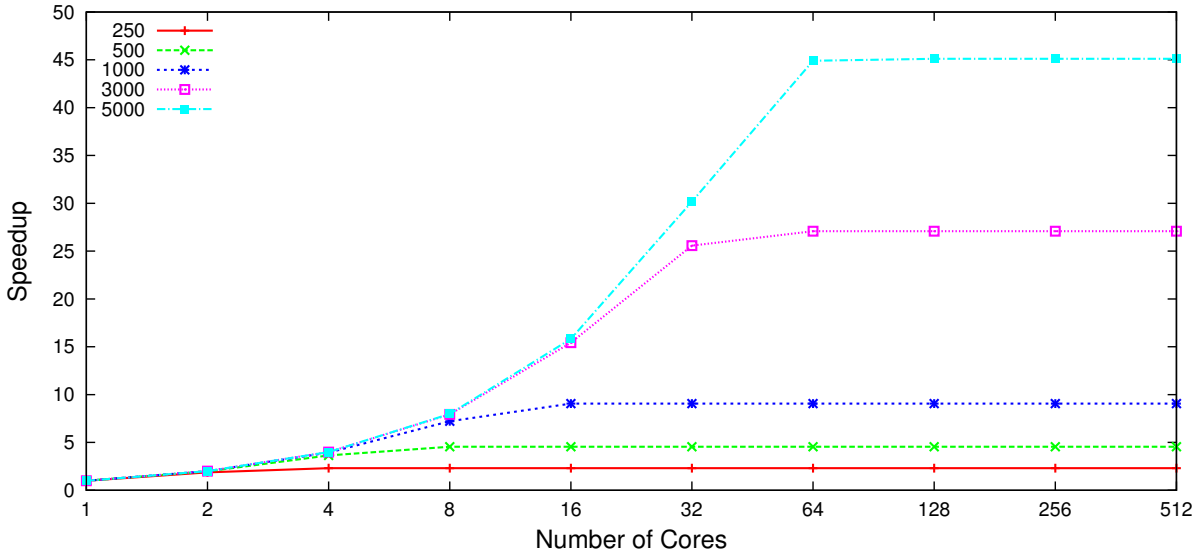


Figure 3.13: The speedup achieved by different multicore systems running Gaussian elimination for different matrix sizes (legend shows matrix dimension).

than a single row (which contains 120 tasks), the processing of non-ready tasks before reaching the next ready task (first task in the second row of Figure 3.7(b)) limits the scalability of this benchmark to at most 8 cores, whereas the benchmark with vertical dependencies in Figure 3.7(c) scales well to 64 cores.

H.264 macroblock decoding tasks have the most complex dependency pattern among the tested benchmarks including the horizontal dependencies between the tasks in the same raw of a video frame. This explains why the limited speedup gain of the H.264 benchmark compared to the independent tasks and the vertically-dependent tasks speedup gains shown in Figure 3.11.

The same benchmarks (Figures 3.7(a, b, and c)) were run on different multicore systems, but with the assumption of a perfect (contention free) memory system. The speedups achieved were almost similar to those depicted in Figure 3.12, and this is mainly due to the dependencies between tasks in every benchmark, not allowing them to reach the point of competing to access memory and thus cause memory contention.

Figure 3.13 shows the speedup achieved by using different multicore systems to solve the Gaussian elimination problem (Figure 3.8) on page 76 for different matrices of sizes ranging from 250×250 to 5000×5000. Memory contention is modeled, and double buffering is used.

Dep. Pattern	Max. Speedup	Max. Scalability
Indep. tasks	54×	64 cores
H.264-like (Figure 3.7 a)	8.2×	16 cores
Hor. deps. (Figure 3.7 b)	8.5×	16 cores
Ver. deps. (Figure 3.7 c)	75×	64 cores
Gaussian 5000 × 5000	45×	64 cores

Table 3.5: Maximum scalability and speedup for the different dependency patterns.

Although the size of the *Kick-Off List* of each entry in the *Dependence Table* is equal to 8, Nexus++ can handle the Gaussian elimination problem for matrices of large sizes. This is because of the dummy entries added to the *Dependence Table*.

As shown in Figure 3.13, the matrix size has a great impact on the speedup gain and the application scalability, since a bigger matrix results in a larger number of tasks of larger granularity. Processing a 5000×5000 matrix scaled up to 64 cores with a speedup of 45×. This experiment includes building and managing a task graph of 12,502,499 tasks with 3,523 FLOPs per task on average as shown in Table 3.3. Each single worker core is assumed to be able to do 2 GFLOPS, which means that the average computation time of each of the aforementioned tasks equals $1.77\mu s$.

Although the 250×250 experiment has very small tasks ($83.5ns$ per task on average), Nexus++ can handle them. The benchmark scaled to 4 cores with a speedup of 2.3×. This demonstrates the applicability of Nexus++ to any kind of applications, even those with very fine-grain tasks. Table 3.5 concludes the maximum number of cores, referred to as maximum scalability in the table, utilized by the different benchmarks to achieve their maximum speedup.

All tables and FIFO lists in the Nexus++ task manager do not exceed 210KB of memory. Nevertheless, they are sufficient to perform all the objectives of Nexus++. For comparison, the Task Superscalar [57], on the other hand, consumes more than 6.5MB. Nexus++ introduces dummy tasks/entries in the *Task Pool* and the *Dependence Table* respectively, uses the *Task Pool* indices as tasks identifiers, and uses its internal structures more dynamically and efficiently, therefore tables' sizes are relatively small.

3.4 Summary

This chapter has introduced Nexus++, a hardware task management accelerator for the StarSs/OmpSs runtime systems. Compared to previous work Nexus++ makes four main contributions. First, it overcomes the limitation that a task can only have a fixed, limited number of inputs/outputs by introducing dummy tasks in the *Task Pool*. It also overcomes the limitation that only a fixed, limited number of tasks can depend on a certain task by introducing dummy entries in the *Kick-Off Lists* of the *Dependence Table*. Second, it support double buffering by integrating a task controller in each worker core. Third, it implements task dependency resolution efficiently, since few hash table lookups are required to determine if tasks depend on each other. Fourth, a platform-independent implementation of Nexus++ is presented, whose parameters are fully configurable.

Experimental results obtained using a SystemC model show that Nexus++ achieves a speedup of 54 × /143× with/without modeling memory contention respectively, for a synthetic benchmark that includes all-independent tasks. Furthermore, double buffering increases the scalability of the system.

Eventually, for large (64 cores and more) systems, the speedup gain starts to decrease, mainly because the application does not exhibit sufficient task-level parallelism, insufficient memory bandwidth, and/or because the master core cannot generate tasks fast enough to keep all worker cores busy. Nevertheless, using a hardware task manager showed its feasibility for improving the scalability of applications with complex task graph.

Furthermore, it is also shown that a benchmark modeled after Gaussian elimination, where the number of tasks that depend on a certain task is not constant and can become very large, ran successfully with a speedup of 45× for a 5000 × 5000 matrix using 64 cores. Even for small matrices, where the resulting tasks are fine-grain, Nexus++ handles them successfully and the benchmark achieved 2.3× speedup using 4 cores.

Although Nexus++ targets StarSs/OmpSs applications, parts of it can be reused for other programming models. For example, it contains hardware queues that can be used for low-latency retrieval of independent tasks.

4 An Integrated Hardware-Software Approach to Task Graph Management

The SystemC model of Nexus++ presented in Chapter 3 shows the importance of hardware support for task graph management. It also presents a design space exploration that gives an idea of the sizes of the internal structures of Nexus++.

In SystemC, however, and as has been presented in the previous chapter, the developer should take care of all the timing details and define them manually such as on-chip and off-chip memory access times. Moreover, the SystemC developer is not able to know whether the design can be synthesized into real hardware or not, and what would be the maximum clock frequency that can run the system.

Therefore, the next step is to implement Nexus++ in a hardware description language such as VHDL. Then test its synthesizability and its hardware characteristics, in addition to evaluating it with micro-benchmarks as well as real applications.

This will give a realistic insight about the efficiency of Nexus++, especially with fine-grain tasks and tasks with complex dependency patterns, which are the goals of this thesis.

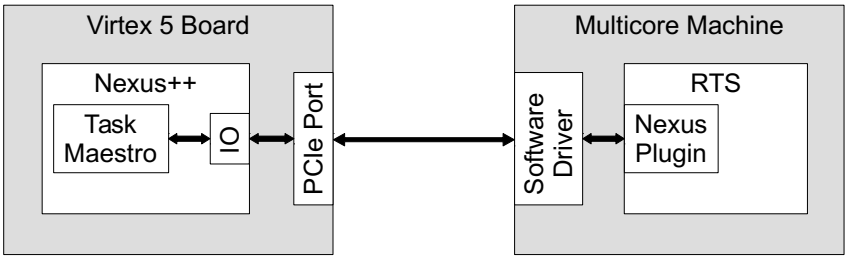


Figure 4.1: Nexus++ high level system overview.

4.1 Design Overview

Nexus++ is thought to be integrated with real multicore systems. Therefore, Nexus++ is implemented in VHDL using the Xilinx XUPV5-LX110T [170] FPGA development board, which is a feature-rich Virtex 5 general purpose evaluation and development FPGA board with on-board memory and industry standard connectivity interfaces such as the PCIe bus. The high level design is depicted in Figure 4.1. The Nexus++ task manager resides on the FPGA, and communicates with the runtime system on the multicore system using the PCIe bus.

To integrate Nexus++ in a generic multicore system:

- Nexus++ has to be implemented, synthesized, and realized on the FPGA board,
- a communication interface that utilizes the PCIe port on the FPGA and exchanges data with Nexus++ has to be implemented,
- a software driver that enables data flow between the runtime system and the FPGA using the PCIe bus has to be developed,
- the runtime system has to be modified in order to replace its current task graph management mechanism by a communication unit (*Nexus Plugin*) with Nexus++,

The FPGA board can then be plugged into the host multicore machine and the task graph management responsibilities can be offloaded to Nexus++.

4.1.1 Functional Overview

Figure 4.2 shows the block diagram of the proposed task manager. It is mainly composed of two units; Nexus input/output unit (*Nexus IO*) which handles communication with the host runtime system, and the task management unit known as the *Task Maestro*, which manages the task graph at runtime and issues tasks when they are ready.

The multicore system under consideration is assumed to have one *Master Core* that executes the main thread and creates *Task Descriptors*, and several worker cores that execute the tasks. A *Task Descriptor* contains task-related information such as its function pointer and input/output list. Nexus++ is responsible for task graph management carried out by the runtime system.

As shown in Figure 4.1, data communication occurs between Nexus++ and the runtime system via the *Nexus Plugin*. So when the master thread creates new tasks, the runtime system submits them to Nexus++. Nexus++ sends ready tasks IDs to the runtime system, and whenever a worker core finishes a task, the runtime system notifies Nexus++ of that finished task's ID.

Each FIFO list in the design is generated using Xilinx Coregen v14.4 FIFO Generator 9.3 [171]. We chose to use First-Word Fall-Through FIFOs, which are FIFOs with registered output. This enables the designer to look ahead to the next available word in the list without issuing a read operation. This will save one clock cycle every time a FIFO list is read. The FIFO generator provides an *output valid* flag indicating, as the name says, whether the current output is valid or not. The generated FIFOs also include *fifo empty* and *fifo full* flags, which help the designer to stall reading/writing whenever the FIFO empty/full flags are set respectively.

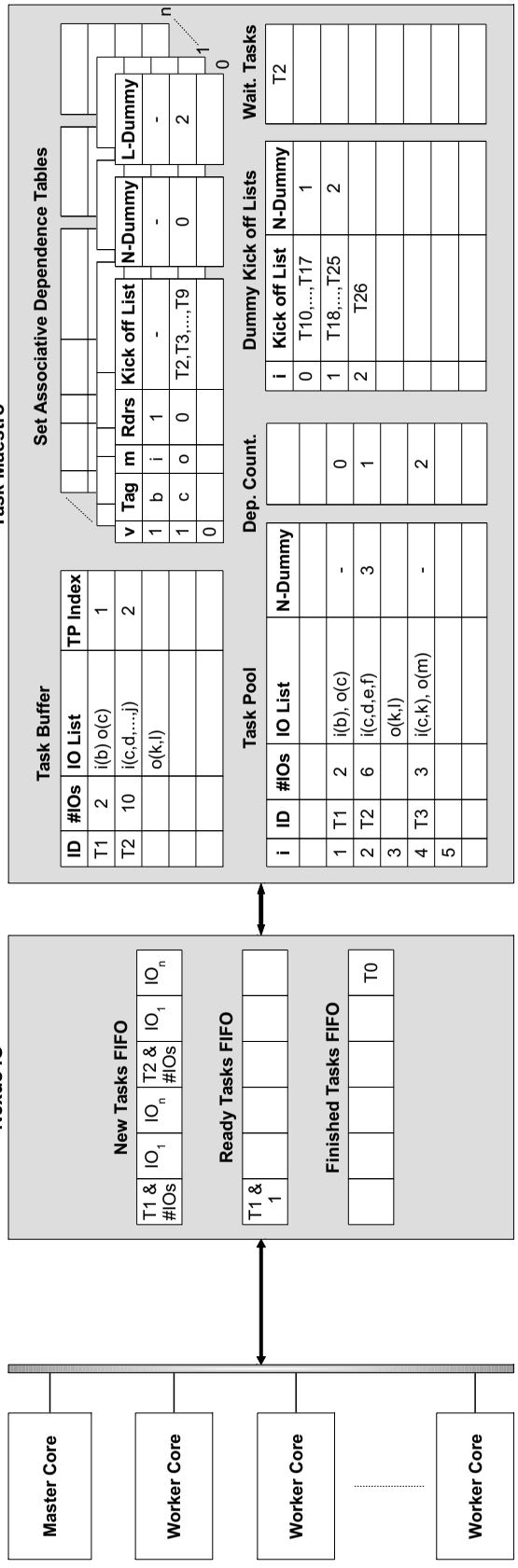


Figure 4.2: Nexus++ block diagram in a multicore system.

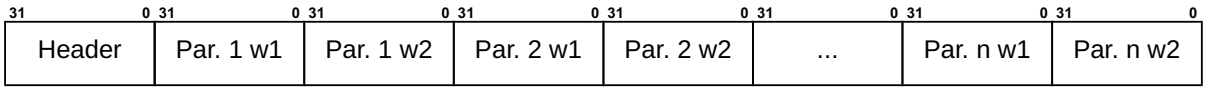


Figure 4.3: The task format as submitted to the *Nexus IO* unit.

The *Nexus IO* unit is designed to communicate with the PCIe port at the FPGA board. Xilinx provides an integrated endpoint for PCIe designs compliant with the *PCI Express Base 1.1* specification, along with an example design that supports single double word (32-bit) payload read and write PCIe transactions [169].

The runtime system submits new tasks to the *Nexus IO* unit shown in Figure 4.2 as 32-bit packets (PCIe payload width), which stores them as 32-bit words in the *New Tasks FIFO* list. Checking the *New Tasks FIFO* list's full flag before submitting new tasks, the runtime system stalls when necessary. Each task starts with a header word containing its function pointer and the number of inputs/outputs, and later words containing task's inputs/outputs as shown in Figure 4.3.

The *Task Maestro* then reads the *New Tasks FIFO* list, generates *Task Descriptors* and stores them in (1) a temporary task storage table called the *Task Buffer*, and (2) in the *Task Pool*, the main task storage table in Nexus++, where tasks reside until the end of their life cycles. The *Task Pool* index at which a certain *Task Descriptor* is stored becomes the unique identifier of this task inside Nexus++. This is an efficient way for reading the *Task Pool*, since no address lookup is required at all.

The *Task Buffer* and *Task Pool* are shown in Figure 4.2. The *Task Buffer* is relatively small (8 - 16 tasks) and is important, along with the *New Tasks FIFO* list, to decouple task submission from processing new tasks. Both tables are implemented as dual-port block RAMs. One port is accessed by the process writing new tasks to them. The second port of the *Task Buffer* is accessed when processing new tasks, and this happens in order. That is why this table is of a small size and tasks' data is not preserved after processing it. The *Task Pool*, on the other hand, is much larger (256 - 1024 tasks), and its second port is accessed arbitrarily upon processing finished tasks. Having three processes accessing the dual-port tables, the *Task Buffer* cannot be merged in the *Task Pool*.

For a certain Task *T1*, the *Task Pool* stores the task's ID in the *ID* field, the number of inputs/outputs in the *#IOs* field, the list of inputs/outputs in the *IO List* field, and a pointer to the next dummy task if the task has a large number of inputs/outputs that do not fit in a single entry in the *Task Pool*. The *IO List* includes the addresses and the mode of access of the task to those addresses. *T1* for example has one input *b* and one output *c*. The *Dependence Table* and the other structures in the *Task Maestro* are described in later sections.

The *Task Maestro* reads incoming tasks one by one from the *Task Buffer*, builds up the task graph and calculates the *Dependence Counter* for the task in progress. This is done by comparing every single input/output of the new task against all inputs/outputs of all previously submitted tasks. The resulting *Dependence Counter*, if greater than 0, is stored in the *Dependence Counts* table shown in Figure 4.2. Otherwise, the task is ready to run and the *Task Maestro* writes its function pointer along with its *Task Pool* index to the *Ready Tasks FIFO* list inside the *Nexus IO* unit. The runtime system polls the latter list's *valid* flag, reads it whenever it has valid data, and schedules ready tasks to run.

Whenever a task is finished, the runtime system communicates only its *Task Pool* index back to the *Nexus IO*, which writes the incoming data to the *Finished Tasks FIFO* list. After that, the *Task Maestro* reads the latter list for the index of the finished task in *Task Pool*, then reads the finished task info from the *Task Pool* and updates the task graph, and finally deletes the finished task entry from the *Task Pool*.

Each one of the tables shown in Figure 4.2, namely the *Task Pool*, *Task Buffer*, *Dependence Counts*, *Dependence Table*, and the *Dummy Kick off Lists* table, is implemented using one or more dual-port RAMs provided by the FPGA, in order to enable concurrent accesses to any of the tables.

The detailed data/control flow inside Nexus++ is described in Chapter 3. There, a *Task Controller* per worker core is responsible for double buffering and communicating with Nexus++. In the new design described in this chapter, those *Task Controllers* are to be implemented as part of the *Nexus Plugin* of the runtime system.

As described in Chapter 3, Nexus++ design ensures deadlock-free processing, and utilizes FIFO lists between the different handlers to pipeline them and implement stalls.

In Section 4.1.2, task graph management will be described to highlight the set-associative dependence table used to store tasks inter-dependence information.

4.1.2 Set Associative Dependence Tables

The *Dependence Tables* shown in Figure 4.2 on page 92 are the storage place of the task graph. Every memory location accessed by one or more tasks will have an entry in the *Dependence Table*. Every time a new memory location is submitted (as an input/output of a task) to Nexus++, the *Task Maestro* searches the *Dependence Table* for this memory location. If it was not found, the *Task Maestro* inserts the new memory location to the dependence table (by writing the *Tag* and access mode *m* fields).

In Chapter 3, the dependence table is a hash table with a simple hash collisions resolution algorithm shown in Equation 3.1. There, each memory location can map to one location in the *Dependence Table*. Therefore, a linked-list structure is implemented as the hash collision resolution mechanism. This implies that searching the *Dependence Table* for a certain memory address can include multiple accesses until finding the correct entry in the linked-list structure that is formed when a collision occurs. Moreover, if a certain memory location was accessed for the first time (i.e., it will be assigned a new entry in the *Dependence Table*), but its hash address in the *Dependence Table* has a long linked-list of entries, searching for this new address includes accessing the *Dependence Table* many times until reaching the end of the list, with ultimately a negative search result. For this reason, the VHDL Nexus++ introduces the *Set-Associative Dependence Tables*, a cache-like structure for maintaining the task graph.

When inserting a memory location in the new *Dependence Table* shown in Figure 4.2 on page 92, it can be stored in one of the *n*-way structure. If all *n* locations, which a certain memory address *A* maps to, are full, then *A* has to wait and the *Task Maestro* stalls. Unlike the *Dependence Table* in Chapter 3, searching the n-way set-associative table for a certain memory address costs only one read operation of the different ways. Comparing the valid*(v)* and *Tag* fields will determine whether the searched memory address is found or not.

Dependency resolution is performed by maintaining a *Kick-Off List* for each memory address. A *Kick-Off List* of a memory address has room for 8 tasks. It records for

each task valid and access mode flags, in addition to the task's *TP index*. There is a counter per *Kick-Off List* in addition to pointers to the head and tail of the list. When for example task *T2* is submitted to Nexus++, its input/output list will be processed one by one. Task *T2* has 4 inputs and 2 outputs as shown in the *Task Pool* in Figure 4.2. When searching the *Dependence Table* for *T2's* first input, namely memory address *c*, the *Task Maestro* will find that *c* was previously inserted to the *Dependence Table* as an output to an older task. Therefore, *T2* will be added to the *Kick-Off List* of *c*, and the *Dependence Counter* of *T2* will be incremented once.

On the other hand, if *c* was inserted as input to an older task, and since that *c* is also input to *T2*, then only the readers count *Rdrs* field in the *Dependence Table* will be incremented, without changing the *Dependence Counter*. After processing all memory pointers in the input/output list of the new task, if the resulting *Dependence Counter* is 0, then this task is ready and will be written in the *Ready Tasks FIFO* list.

Whenever a task finishes executing, its input/output list will be fetched from the *Task Pool* and processed. For example, when task *T1* finishes, its input/output list is looked up in the *Dependence Table*. Memory address *b* has an empty *Kick-Off List*, and therefore can be invalidated. Memory address *c*, on the other hand, has some tasks in its *Kick-Off List*.The *Task Maestro* reads the first task (*T2*) in this *Kick-Off List* and decrements its *Dependence Counter*. If the resulting *Dependence Counter* equals 0, then task *T2* will be sent to the *Ready Tasks FIFO* list. Finally, depending on whether *T2* is reading-only or writing memory address *c*, the *Task Maestro* decides to further read tasks from the *Kick-Off List(c)* or not.

The short example above shows how Nexus++ handles read-after-write dependencies. Nexus++ handles also write-after-read and write-after-write hazards (although these two are name dependencies) as described in Chapter 3.

4.1.3 Dummy Tasks and Entries

The dummy tasks and entries mechanism that is explained in Chapter 3, is also implemented in the VHDL version of Nexus++. Tasks that cannot fit into one entry in the *Task Pool* will simply be stored in multiple entries, which are linked together using the *N-Dummy* field in the *Task Pool*, as shown in Figure 4.2 on page 92.

For example, task *T2* shown in the *Task Pool* in Figure 4.2 has 6 memory locations in its input/output list. Since in our design, each entry in the *Task Pool* can have up to 4 inputs/outputs, the 2 extra parameters are stored in another entry (at TP(3)) in the *Task Pool*, and a pointer to TP(3) is inserted in the next dummy (*N-Dummy*) field of TP(2) where the first 4 parameters reside. Although the number of inputs/outputs per entry in the *Task Pool* is determined to be 8 in the SystemC prototype of Nexus++ presented in Chapter 3, it is set to 4 in the VHDL prototype presented in this chapter in order to reduce the size of Nexus++. This has been decided after profiling several benchmarks as described in Section 4.3.1 and finding out that most of them have less than 4 parameters. Only one benchmark has few tasks that have up to 6 parameters, the case that Nexus++ handles by inserting dummy tasks in the *Task Pool* as described here.

Although this solves the problem of having a fixed, limited number of inputs/outputs per task, the maximum number of inputs/outputs is still bounded by the size of the *Task Pool*.

The same principle is deployed in the *Dependence Table*, where the *Kick-Off List* has a limited size of 8, thus restricting the number of tasks that might depend on a certain memory segment. An example is shown in the *Dependence Table* in Figure 4.2 too. Memory address *c* has more than 8 tasks waiting for it. The first 8 tasks are recorded in the direct *Kick-Off List* of *c*, and the extra ones are recorded in an additional table (the *Dummy Kick off Lists* shown in Figure 4.2) specially created to handle this scenario.

Two pointers to the dummy *Kick-Off List(s)* are recorded in the original *Dependence Table*: (1) the next dummy (*N-Dummy*) pointer points to the immediate following *Kick-Off List*, and (2) the last dummy (*L-Dummy*) pointer points to the last dummy *Kick-Off List* that might still have some room for more tasks, indicating where should the next waiting tasks be added to. The *N-Dummy* pointer is important when the *Kick-Off List(c)* is to be read, since reading *Kick-Off Lists* should be performed in a first-in first-out order.

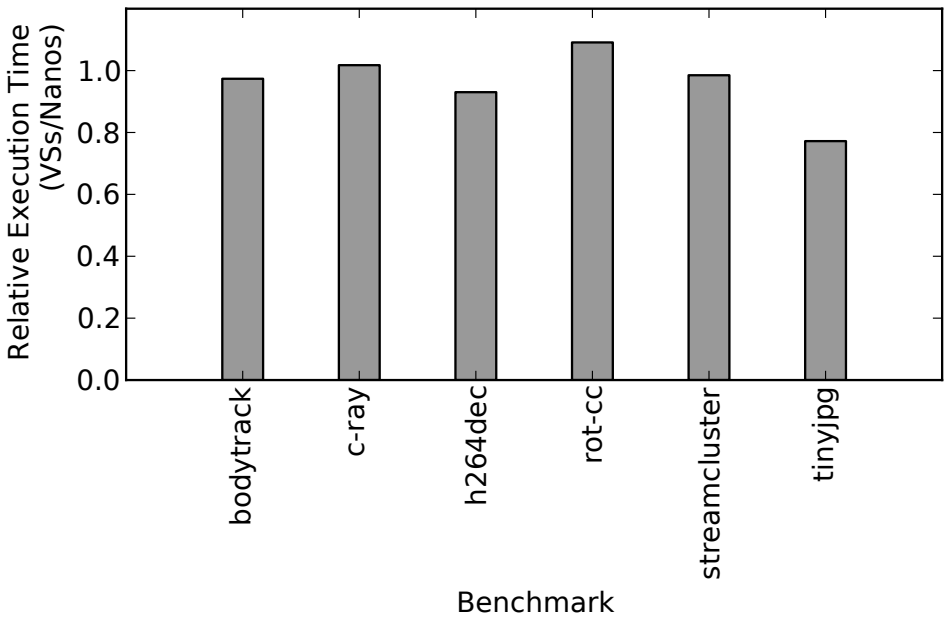


Figure 4.4: Execution time comparison between Nanos and VSs, for various benchmarks on a 4-core machine.

4.2 System Integration

4.2.1 VSs Runtime

VSs is another research project at the Embedded Systems Architecture group at TU Berlin. It is based on a proto-runtime system that provides basic management of tasks: creation, switching and destruction. It has been used to implement the functionality of OmpSs including dependency resolution, synchronization primitives such as *Taskwait* and *Taskwait on (*mem_addr)*, as well as a simple scheduling policy. Having been developed in the same group, VSs has the advantage over Nanos of easier debugging and faster integration.

VSs is used in order to test Nexus++ with real applications as depicted in Figure 4.1 on page 90. VSs runtime library provides the same API as the Nanos runtime [54] (official OmpSs runtime system). It uses OmpSs's source-to-source compiler, called Mercurium, which transforms pragmas into function calls to the Nanos runtime library, but replaces Nanos, in order to support exchanging tasks' information with Nexus++.

As shown in Figure 4.4, the performance of VSs compared to Nanos is very similar for a set of real benchmarks. In fact, VSs is slightly faster than Nanos for most of the

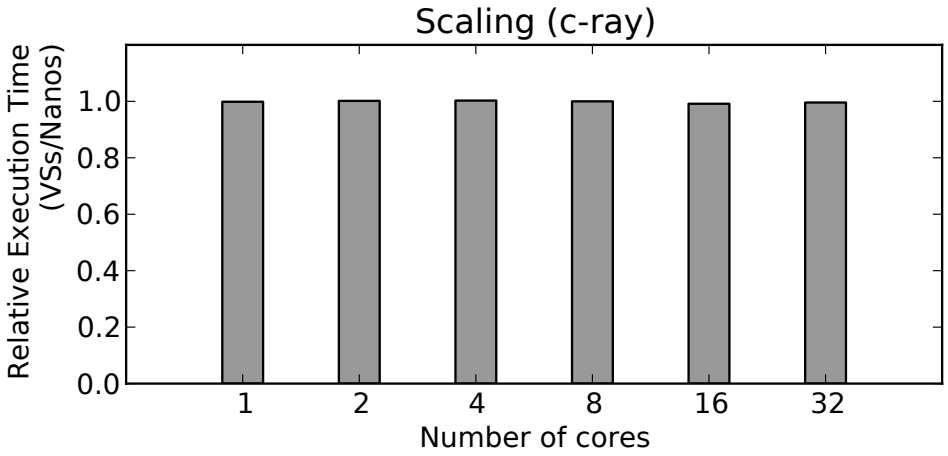


Figure 4.5: Scaling behavior comparison between Nanos and VSs, for the *c-ray* benchmark.

benchmarks. This indicates that if the proposed hardware accelerator can overcome the scalability bottleneck of VSs, then it can also be beneficial for Nanos.

Figure 4.5 compares the scalability behavior of VSs and Nanos, by running the *c-ray* benchmark which has independent tasks only. Figures 4.4 and 4.5 show that VSs is very similar to Nanos in terms of performance and scalability.

4.2.2 Nexus++ API

VSs provides a simple API to program and communicate with Nexus++. This API abstracts Nexus++ to the programmer and makes it easy to use.

The *Nexus IO* unit shown in Figure 4.2 on page 92 provides three FIFO lists to communicate with the host runtime system VSs. Every new task will be submitted to the *New Tasks FIFO*, available ready tasks can be read from the *Ready Tasks FIFO*, and finally when a task is completed, the VSs runtime notifies Nexus++ by writing the finished task ID to the *Finished Tasks FIFO*. Those FIFO lists have also status registers that can be read by the runtime in order to make sure that a certain FIFO list has enough room before writing it, or has some data before reading it.

The main operations provided by the software API are:

- `void VSs_init_nexus();`
  This method is used to map Nexus++ I/O unit in the system's memory space, in addition to initializing the different structures inside VSs and Nexus++.

- `void VSs_cleanup_nexus();`
  This method resets Nexus++ in addition to all the structures inside VSs related to Nexus++.
- `void VSs_submit_task_to_nexus(* task_descriptor);`
  This method is used to submit a new task to Nexus++, by writing it to the memory-mapped space of the *New Tasks FIFO* list in Nexus++ I/O unit shown in Figure 4.2 on page 92.
- `void VSs_end_task_to_nexus(* task_descriptor);`
  This method is used to submit a finished task to Nexus++, by writing its ID to the memory-mapped space of the *Finished Tasks FIFO* list in Nexus++ I/O unit in Figure 4.2.
- `void VSs_get_ready_tasks_from_nexus();`
  This method is used to read a ready task ID from Nexus++, by reading the memory-mapped space of the *Ready Tasks FIFO* list in Nexus++ I/O unit in Figure 4.2.
- `int VSs_nexus_has_outstanding_tasks();`
  This method can be used to check how many tasks are still stored inside Nexus++.
- `int can_write_infifo();`
  This method can be used to check whether the *New Tasks FIFO* list in Nexus++ has enough empty space to write a new task descriptor into it. This method is mainly used by the `VSs_submit_task_to_nexus(task_descriptor)` method.
- `int can_write_finfifo();`
  This method can be used to check whether the *Finished Tasks FIFO* list in Nexus++ has enough empty space to write a finished task ID into it. This method is mainly used by the `VSs_end_task_to_nexus(task_descriptor)` method.
- `int can_read_outfifo();`
  This method can used to check whether the *Ready Tasks FIFO* list in Nexus++ has new ready tasks to read them by the runtime system and schedule them to run consequently.

	# tasks	total work (ms)	avg task size (μs)	# deps
c-ray	1200	7381	6151	1
emptytask	1000	1	1	2
h264dec	57051	833	15	2-6
rot-cc	16262	8150	501	1
sparselu	54814	38128	696	1-3
streamcluster	652776	237908	364	1-3

Table 4.1: Overview of the benchmarks. Work durations obtained from traces collected on Xeon E7-4870.

4.3 Experimental Setup

To evaluate the performance of Nexus++, several benchmarks are used based on a series of trace-based simulations as well as by executing some real benchmarks using the Nexus++ FPGA implementation on a PCIe extension board.

The goal of the evaluation is to know the impact of Nexus++ on the performance of the different benchmarks, especially those with fine-grain tasks, or with complex dependency patterns.

4.3.1 Benchmarks

Several benchmarks from the Starbench benchmark suite [6] are used to evaluate Nexus++. These include *c-ray* (ray tracing), *h264dec* (H.264 video decoding), *rot-cc* (image rotation and color conversion) and *streamcluster* (k-median clustering). In addition, two other benchmarks are tested: the *emptytask* (a synthetic benchmark) and the *sparselu* (sparse LU matrix factorization).

The chosen benchmarks range from totally parallel workloads to workloads with more complex dependency patterns.

c-ray and *rot-cc* have simple dependency patterns, with tasks working on each line of an input image independently. For *c-ray*, there is only one task per line, which means that all tasks are independent. For *rot-cc* there are two tasks per line, one for rotation and one for color conversion, with the second depending on the first. All pairs are independent of each other. *c-ray* is the best case as it has large tasks and ample parallelism, thus most runtime overhead can overlap with task execution.

emptytask is a synthetic benchmark consisting of a sequence of 1000 tasks of minimal task size (one addition operation) each dependent om the previous. In effect, this represents a worst case, as there is no parallelism in the application at all, and runtime overhead cannot be hidden behind task duration either. Together with *c-ray* this benchmark gives an idea of the performance range of the different runtimes.

streamcluster is a streaming data analysis kernel with fork-join-style parallelism. It consists of a chain of groups of about 400 tasks followed by a `taskwait`.

sparselu and *h264dec* have more complex dependency patterns. *sparselu* is a sparse matrix LU factorization kernel from the developers of OmpSs. It scales well, as the granularity is designed to match Nanos overheads. The H.264 decoder, on the other hand, has small tasks with many dependencies. This fine-grain parallelism is especially challenging to manage as introduced in Section 2.2. It is possible to adjust the task size to expose more parallelism, but at the cost of increased overhead as more tasks need to be managed. The granularity chosen for these experiments (2×2 macroblocks per task) was determined based on Andersch et al. [6] to produce fine-grain tasks, finer that the granularity (8×8 macroblocks per task) suggested by the authors, in order to exploit more parallelism in the application as introduced in Section 2.2 and to stress-test the different runtime systems (Nanos, VSs, Nexus++).

Table 4.1 summarizes the relevant characteristics of the benchmarks in focus. Using those benchmarks, several experiments were performed, ranging from trace-based cycle-accurate simulations of these benchmarks, to truly running them on a multicore machine.

4.3.2 VSs-Nexus++ Setup

The Nexus++ design described in Section 4.1.1 is implemented on a Xilinx XUPV5-LX110T development board. The board is then installed in a machine equipped with an Intel Core i5-2500K 4-core CPU due to the lack of physical access to machines with larger number of cores.

Communication takes place over the PCIe bus. Delays of the read and write operations are measured using VSs: writing one word over the PCIe bus consumes approximately 250ns in this experimental setup, and reading one word consumes 400ns.

The width of a parameter of a task is assumed to be 48[1] bits, the maximum virtual address space size in the x86-64 architecture [80]. Therefore, for a task with n arguments, $(n + 1)$ words are sent to Nexus++ (n words for the task's parameters, and 1 header word), followed by reading 1 word from the *Ready Tasks FIFO* when the task becomes ready, and then writing a finished task notification of 1 word after the task is completed. So in total, $n + 2$ words are written and 1 word is read for a task with n arguments. Practically, the communication time is more than that solely needed to communicate the *Task Descriptor*, since in order to to communicate with the current Nexus++ implementation, the runtime also needs to read the status registers to ensure flow control, and change the endianness to conform to the PCIe protocol. These steps incur additional overhead.

VSs is implemented to run in three operational modes:

1. the software-only mode, where it computes the task graph and schedules tasks on its own,
2. the VSs-Nexus++ mode, where VSs sends the *Task Descriptors* to Nexus++ and waits for the latter to compute the ready tasks IDs and returns them to VSs,
3. the debugging-mode, where VSs submits tasks to Nexus++, and at the same time computes the task graph as in the first mode in order to compare the results to those of Nexus++.

In the VSs-Nexus++ and debugging modes, communication is implemented using memory-mapped I/O over the PCIe bus.

4.3.3 Simulation-based Setup

Traces of the aforementioned benchmarks were collected on a 40-core Xeon E7-4870 machine running at 2.40GHz.

These traces include the task descriptors, which specify the inter-task dependencies, and the execution time of each task. Using the information from the traces, three sets of simulations were performed:

[1]Bits 63:48 are a sign extension of bit 47.

1. No Overhead: In order to determine the lower bound for the execution time of the benchmarks, the execution of an application without any overhead is simulated. In this simulation, the simulation time does not advance while dependencies are being resolved. Only the execution time of the tasks is taken into account. This allows us to determine when the lack of available parallelism in the application is the limiting factor.
2. Nexus++ only: This simulation additionally accounts for the dependency resolution overhead incurred by the Nexus++ core. Failure to scale in this simulation indicates a bottleneck inside the design. In this simulation free worker cores start executing tasks directly after they are reported as ready by Nexus++. No communication or other non-dependency resolution overhead is accounted for.
3. Nexus++ and runtime: Here, an additional delay of $(n + 1) * 250 + 400$ns is introduced between Nexus++ reporting a task as ready and the start of execution of the task by the worker core, as well as a delay of 250ns between the end of the task and the reception of the finished task notification by Nexus++. This represents the overhead of communication between the processor cores and Nexus++, as described in Section 4.3.2. Although the setup in Section 4.3.2 (an Intel Core i5-2500K machine) is different from the simulation setup (a Xeon E7-4870 machine) in this section, the PCIe interface is a standard bus and the communication latencies from the Intel i5-2500K machine can be reflected on the Xeon E7-4870 machine for evaluation purposes.

Additionally, we measured the overhead due to runtime features that Nexus++ does not replace (such as setting up the stack for the task and switching execution to it) in VSs to be approximately 5μs per task. In this simulation, task length is increased by this constant, to account for all necessary parts of execution.

The VHDL testbench is set up to run Nexus++ at 12.5MHz and the *Nexus IO* FIFOs at 100MHz, to match the speeds reported by the synthesis tool of the FPGA implementation of Nexus++.

These simulations were compared to the actual runs of the benchmarks on the same machine that the traces were collected on, compiled using the Mercurium compiler version 1.3.5.8 and linked to the accompanying Nanos runtime library.

Of special note is the h264dec benchmark, which is the only benchmark to use the `#pragma taskwait on (*p)` construct that is not supported by Nexus++. A `taskwait` instructs the issuing thread to suspend until all child tasks have finished execution, and `taskwait on (*p)` requires to wait only for those child tasks that access memory address `p`. It is thus functionally correct to replace instances of `taskwait on (*p)` with `taskwait`. However, this will decrease the available parallelism of the application. In the following experiemnts, this is how VSs handles `taskwait on (*p)` when running with Nexus++. We report both the *No Overhead* simulation with `taskwait on (*p)`, which represents the parallelism available when executing with the Nanos runtime library, as well as the *No Overhead* simulation with `taskwait on (*p)` replaced by `taskwait`, which is the parallelism that remains available to Nexus++.

4.4 Evaluation Results

4.4.1 Simulation Results

Figure 4.6 shows the scaling behavior of the three simulations, as well as the Nanos-linked run for comparison. Performance is reported as speedups relative to the *No Overhead* simulation on one core (origin of the solid red line), which corresponds to the total amount of work in the application.

For three of the four benchmarks with sufficient available parallelism to scale linearly to 32 cores (*c-ray*, *rot-cc* and *sparselu*), Nexus++ itself adds negligible overheads in all cases. In comparison, Nanos adds a slight overhead in the cases of *c-ray* and *rot-cc*, but has a significant overhead for the benchmarks with complex inter-task dependencies (*sparselu*).

The *emptytask* benchmark allows us to evaluate the response to lack of parallelism, as well as estimate the overhead in absolute terms. In this benchmark, only a single task is ready at a time, so additional cores cannot improve the performance, but should avoid degrading it. Both Nexus++ and Nanos deal with this case well. In absolute numbers, an average overhead per task of 3μs for Nexus++ and 10μs for Nanos is observed.

The final two benchmarks, *h264dec* and *streamcluster*, are applications with limited parallelism. For the H.264 decoder, using the `taskwait on (*p)` construct, a speedup of up to 13× can theoretically be achieved (solid red line). The Nanos runtime, however, is incapable of using this parallelism, achieving less than sequential performance no matter how many cores it uses.

Because it does not support `taskwait on (*p)` and instead replaces it with `taskwait`, the parallelism available to Nexus++ is bounded at 3.5× speedup (dashed yellow line). Even with this limitation, Nexus++ performs better than Nanos, achieving 2.8× speedup. This shows that for fine-grain tasks, hardware task management is of great benefit.

The task size in *h264dec* was chosen to exploit much parallelism in the application. In this benchmark 2×2 macroblocks are processed per task. With a more efficient runtime, it would be possible to decrease task granularity, down to a single macroblock per task.

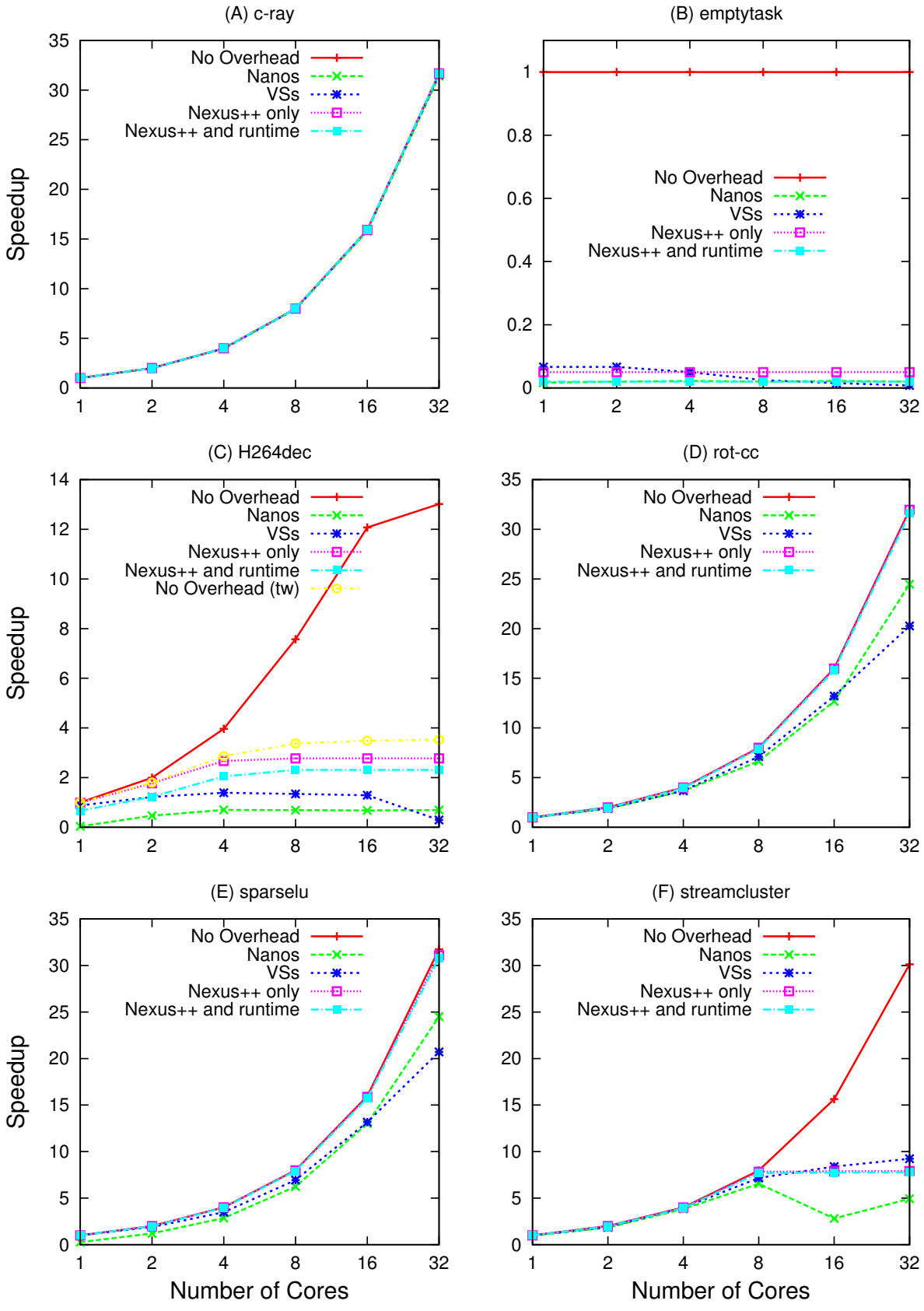


Figure 4.6: Scaling behavior of the benchmarks under consideration.

Benchmark	Speedup
c-ray	1-1.01×
emptytask	0.9-1.2×
h264dec	2.61-16.75×
rot-cc	1.02-1.29×
sparselu	1.21-3.75×
streamcluster	1.01-2.75×

Table 4.2: the speedups gained from *Nexus++ and runtime* compared to Nanos runtime system.

Streamcluster in theory continues to scale linearly beyond 8 cores. In practice, some bottlenecks intervene to limit scaling at 8× for Nexus++, and performance degrades even worse for Nanos when run on more than 4 cores. One peculiarity of the *streamcluster* benchmark is that it has many tasks demanding read access to the same memory location, potentially leading to contention of the resources used to manage it. This may be the reason for the failure to scale beyond 8 cores, with either Nanos or Nexus++.

In summary, the speedups gained from *Nexus++ and runtime* compared to Nanos are shown in Table 4.2.

4.4.2 Results using VSs and the Nexus++ FPGA Implementation

To confirm that Nexus++ works correctly in practice, a prototype was developed on a Xilinx XUPV5- LX110T FPGA board and tested with a machine equipped with a Intel Core i5-2500K 4-core CPU as described in Section 4.3.2.

The same benchmarks as before are compiled and linked to the VSs runtime described in Section 4.2.1. Three versions were prepared:

1. Software dependency resolution: This is the baseline against which the Nexus++ FPGA implementation is evaluated.
2. Nexus++: This version uses Nexus++ for dependency resolution with VSs and Nexus++ communicating over the PCIe.
3. Nexus++ and improved flow control: This version also uses Nexus++, but the *Nexus Plugin* in VSs is aware of the size of the *New Tasks FIFO* and the *Finished Tasks FIFO*. Instead of checking for every packet if there is sufficient space, it

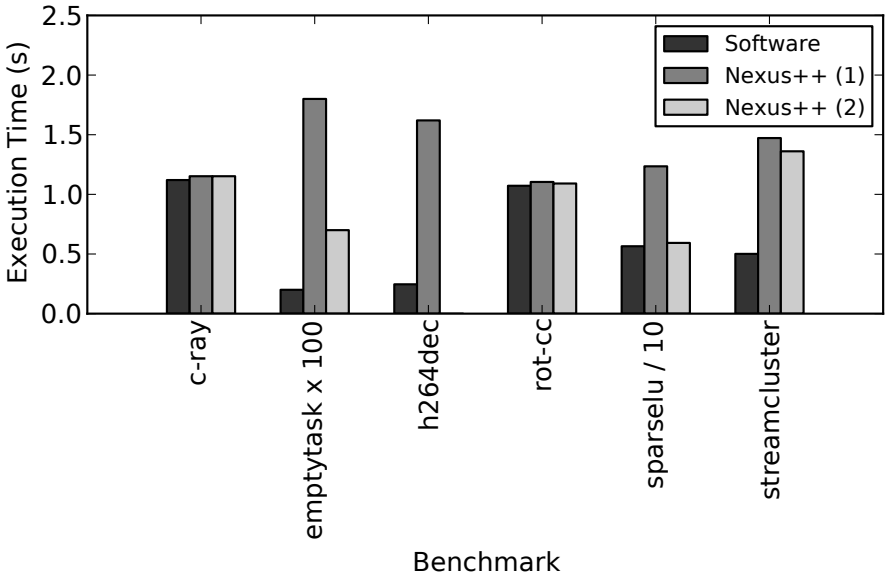


Figure 4.7: Dependency resolution in software vs (1) with Nexus++ and (2) with Nexus++ and improved flow control. In order to fit the graph, the bars for *emptytask* and *sparselu* were scaled by the indicated constant factor in the benchmark name.

will read the FIFO occupancy counter once and keep a conservative estimate of space available. Only when there is a possibility of the FIFO being full will the *Nexus Plugin* check again before writing to either FIFOs.

The results of running the three above-mentioned versions of each of the benchmarks are shown in Figure 4.7. The communication over PCIe shows a significant impact. From the *emptytask* benchmark, we can again derive an absolute estimate of overhead. The naive implementation takes 12.5μs per task. With flow control optimization, the overhead is reduced to 6.9μs per task. In comparison, the software version only takes 1.2μs per task.

Measuring where time is spent inside the *Nexus Plugin* reveals that there are two main sources of overhead. The first, reading the status registers for flow control (which is addressed by the flow control optimization described above). The second is in the nature of the communication protocol: the runtime has to poll the *Ready Tasks FIFO* for new tasks to execute, as Nexus++ is not capable of initiating transfers in this implementation. This second source of overhead is especially important for

benchmarks which have few parallel tasks, and thus higher likelihood of idle cores polling repeatedly and interfering with transfer of task finish notifications.

Accordingly, the results show that for the benchmarks with large amounts of available parallelism such as *sparselu*, optimizing the flow control reduces overhead by 95%. For *streamcluster*, which only spawns up to 20 tasks at a time, improvement is only 11%, as polling overhead dominates. This version with flow control optimization triggers a runtime error in the *h264dec* benchmark, so no results are provided for it.

4.5 Summary

The VHDL implementation of Nexus++ has been presented in this chapter, as well as its integration in real multicore systems as a PCIe extension board using VSs, a runtime library capable of making use of the Nexus++ accelerator.

Evaluating the FPGA prototype of Nexus++ with real applications, the communication protocol between the VSs runtime system and the Nexus++ over the PCIe bus is suboptimal and adds significant delays. Nevertheless, the PCIe extension board is functional and the performance reaches levels similar to the software version after applying flow control optimization. This leads us to conclude that with an improved communication protocol, or even bringing Nexus++ closer to the cores on chip, Nexus++ would significantly speed up dependency resolution for OmpSs applications. One plausible approach would be to tightly couple Nexus++ as an on-chip co-processor for an ARM multicore SoC using the AXI high performance bus.

This chapter presented a solid implementation of Nexus++ that can be integrated in real SoC. It also demonstrated the positive impact of Nexus++ over the performance of applications with fine-grain tasks as well as complex inter-task dependencies.

Furthermore, the implementation of Nexus++ deepened my insight of data and control flow of task management, and opened more ideas to improve it. For example, more OmpSs pragmas can be supported, and the processing pipeline can be parallelized in a distributed fashion as will be presented in the next chapter. The goal is to bring the scalability figures of the benchmarks under consideration closer to the ideal scalability curve.

5 Nexus#: A Distributed Approach to Task Graph Management

In the previous chapter, Nexus++ is presented. It is a hardware accelerator for runtime systems of task-based programming models such as OmpSs. Task graph management responsibilities that are usually handled by the runtime system, are off-loaded to Nexus++. It tracks tasks' input/output information and utilizes simple table lookups to dynamically build the task graph and find ready tasks to run.

The hardware accelerator presents a significant enhancement in terms of scalability for various applications. It is used to speed up the dependency resolution in OmpSs's runtime system. Tasks processed by Nexus++ can have various dependency patterns and it is possible that tasks can have arbitrary number of *input/output* parameters.

The VHDL implementation of Nexus++ presented in Chapter 4 is the first real prototype of the hardware task manager. Therefore, some room is still available to further improve it. Nexus++ uses a single task graph architecture to resolve dependencies, which may limit the expected scalability that can be obtained by Nexus++. Although significant enhancements are introduced by Nexus++, using multiple task graphs to process tasks in parallel can introduce further enhancements in terms of scalability, aiming at achieving maximum possible performance.

Moreover, dependencies between tasks are expressed in OmpSs by several pragmas. Nexus++ has limited support for these pragmas which limits the performance of some applications employing other pragmas, namely the H.264 decoding benchmark, which uses the *taskwait on* barrier pragma.

This chapter presents Nexus#, a hardware accelerator for task-based dataflow programming models in general, and for OmpSs in its current prototype. Nexus# introduces a scalable distributed task graph manager, where multiple tasks can be analyzed in parallel. This further improves the scalability of applications with fine-grain tasks

or complex dependency patterns. It is implemented as a synthesizable VHDL prototype, aiming at on-chip integrability with future multicore SoCs as a co-processor. It supports the *taskwait on* barrier pragma, among others, which can enhance the performance of various applications.

Before presenting the design of Nexus#, the next section discusses the processing pipeline of Nexus++, and highlights the spots where it can be improved, in order to draw the roadmap of Nexus#.

5.1 Nexus++ Processing Pipeline

The Nexus++ task manager presented in previous chapters is a hardware accelerator for runtime systems of task-based programming models such as OmpSs. Task graph management responsibilities that are usually handled by the runtime system, are offloaded to Nexus++. It tracks tasks' input/output information and utilizes simple table lookups to dynamically build the task graph and find out ready tasks to run.

In Chapter 4, the VHDL prototype of Nexus++ is presented, which thoroughly describes the design and implementation as well as a trace-driven evaluation testbench. Also in the same chapter, the integration process with a real multicore runtime system is highlighted, as well as the evaluation of Nexus++ with real applications.

Nexus++ processes the incoming tasks in a pipelined fashion. It has a simple 3-stage pipeline.The pipeline is shown in Figure 5.1 and is an example for processing tasks that have 4 parameters each.

The first stage is the *Input Parser* stage, and it handles receiving the new tasks from the host multicore machine. It makes sure that all the parameters of the new task have been received before forwarding it to the next stage.

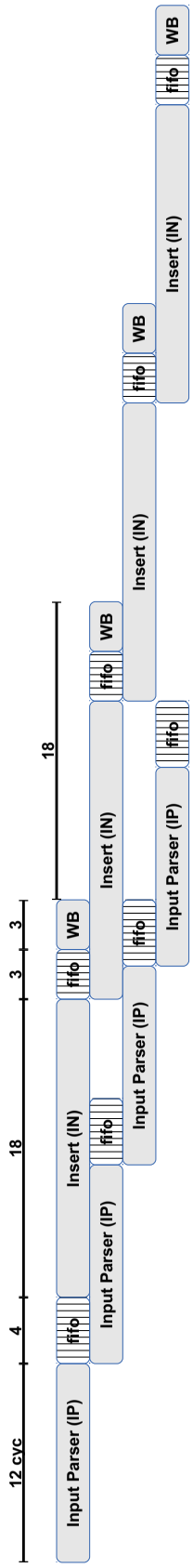


Figure 5.1: Nexus++ Pipeline.

Data communication between the different stages are done using *FIFOs* lists. These lists have status flags, such as *fifo_empty*, *fifo_full*, and *data_valid* flags, which serve as the synchronization signals between the different pipeline stages. The first stage needs two cycles to receive every memory address in the task's input/output list, plus 4 cycles for the header word and synchronization, giving 12 cycles per task. Once the *data_valid* flag of the first *FIFO* list gets activated, the second stage, namely *Insert*, gets triggered. This is the longest stage in the pipeline, which, as the name indicates, handles the insertion of the new tasks into the *Dependence Table*, as elaborated in Chapter 4. This stage needs 18 cycles for our 4-parameter task example.

The result of the *Insert* stage decides whether the third stage will be activated or not. The third stage is the *Write Back* stage, and is responsible for sending ready task IDs back to the *Nexus IO* unit, in order to be read by the host multicore machine subsequently. This means that if the inserted task had dependencies on older tasks, it must wait until its dependencies are fulfilled, and therefore cannot be reported by the *Write Back* stage as a ready task. If the inserted task on the other hand had no dependencies, the *Write Back* stage needs 3 cycles to write it back to the *Nexus IO* unit.

Using the *FIFO* lists as buffers between the different stages introduces extra delays in the pipeline. The delay of the first *FIFO* list in the pipeline can be ignored, since the second stage is longer than the first one, therefore this delay is only effective when the pipeline is empty. On the other hand, the delay of the second *FIFO* is always effective, since the *Write Back* stage is much shorter than the *Insert* stage. This *FIFO* is important to decouple the last two pipeline stages though.

The pipeline shown in Figure 5.1 is, as mentioned before, an example of inserting tasks that have 4 parameters each. In real cases, tasks might have varying number of parameters. This implies that stalls might happen in one or more stages, for example when the task graph has no more room.

There exists a second pipeline responsible for handling finished tasks. Handling finished tasks includes kicking off any waiting tasks, and cleaning up Nexus++ tables by deleting the related information of the finished tasks.

Although Nexus++ has one central task manager, it demonstrated significant improvement over the software runtime system using trace-based simulations. Neverthe-

less, Nexus++ could not improve the scalability of the H264dec benchmark over the software version, since it does not support the barrier pragma *taskwait on*.

Having lots of buffering in its pipeline, in addition to having relatively long as well as variant pipeline stages leave some room for further improvement and optimization, as will be discussed in the following section.

5.2 Nexus#: Distributed Task Graphs

The first thing to notice in the processing pipeline of Nexus++, Figure 5.1, is that the smallest unit processed by Nexus++ is a whole task; i.e., the *Insert* stage does not start before having all task parameters completely buffered. One idea is to parallelize the *Insert* stage in the pipeline, by replicating the task graphs, and distributing the different memory addresses in a task's input/output list among them.

Looking at the original pipeline again, and having Amdahl's law [5] in mind, we notice that parallelizing only the *Insert* stage will yield a maximum of 2× speedup in ideal cases, since the first and third stages in the pipeline are still serial.

Furthermore, the 2× speedup is an ideal situation, since practically parallelizing the insertion process of the different parameters of a task, implies that a scatter-gather process should take place, since those parameters belongs to the same task, and a final decision must be made whether this task is ready or not. This introduces an additional overhead that adds to the serial part in Amdahl's equation.

Moreover, the pipeline under consideration is one example of tasks that have 4 parameters each. In real applications, tasks might have varying number of parameters. In fact, the set of benchmarks which are used have in most cases up to 3 parameters, and in only one case (h264 decoding) 2 to 6 parameters. This implies that the maximum task graphs that can be practically used equals the maximum number of parameters a task can have. Which is a scalability hard upper limit. Furthermore, whenever a task with only one parameter is to be inserted, only one task graph will be busy, and the others will be idle.

In order to overcome the above limitations, two design decisions have been made.

1. The first stage of the pipeline must be broken down into smaller steps.

2. Not only parallelizing the insertion process of parameters of a single task is to be implemented, but also those from different tasks.

The latter decision is to ensure that applications with very few number of parameters per task can also utilize the different task graphs simultaneously, and thus removing the upper limit on the number of task graphs that can be used, at least for this obvious scenario.

5.2.1 Nexus# Design Overview

The block diagram of the proposed distributed task graph system, named Nexus# is depicted in Figure 5.2. Looking at Figure 5.2 from top-to-bottom, tasks are submitted to the *Nexus IO* unit. It has the same interface as in Nexus++, which is necessary to comply with communication protocol between the runtime system and Nexus++ used in Chapter 4.

Since the idea behind Nexus# is to parallelize the insertion process of tasks' parameters using distributed task graphs, and since the different parameters of a single task might go to different task graphs, a scatter-gather approach must be implemented.

One might think at first that whole tasks should be distributed instead to avoid the *gathering* step, but this way dependencies between tasks cannot be tracked, which contradicts the main functional requirement of the hardware co-processor.

5.2.2 Input Parsing

Since the *Input Parsing* stage in Nexus++ pipeline (Figure 5.1) waits for a whole task to arrive before forwarding it to the next stage, this stage is relatively long, and as described before will be a scalability bottleneck according to Amdahl's law if left as is. Therefore, the *Input Parser* in Nexus# reads new tasks' parameters from the *Nexus IO*, and distributes them immediately among the different task graphs shown in Figure 5.2. This way, the insertion process of the first parameter of a task can start, even before the second or rest of parameters of the same task have arrived. Furthermore, parameters of different tasks can be inserted in parallel, as long as they do not share the same task graph.

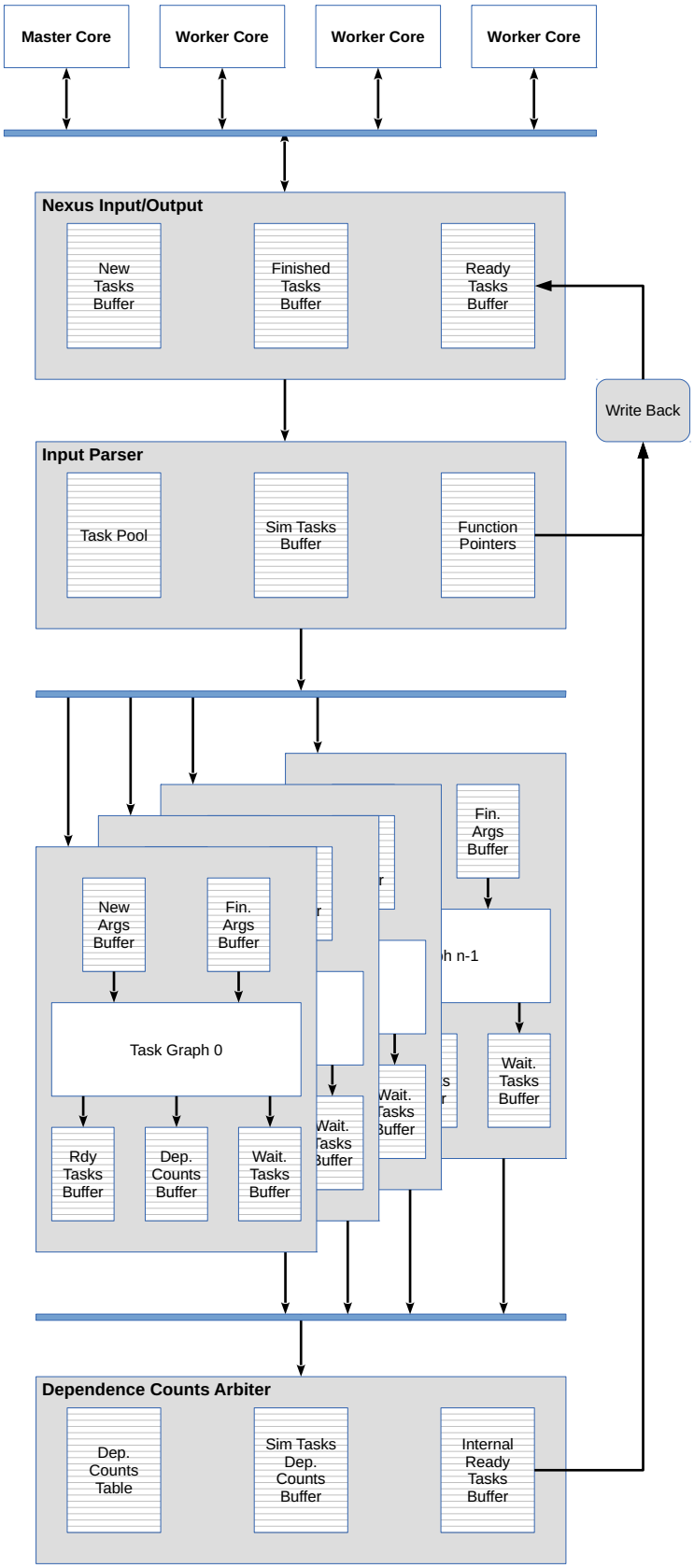


Figure 5.2: Nexus# block diagram.

A key to enhanced utilization and scalability of Nexus# is the distribution algorithm. It should have two essential properties; speed and fairness. Speed, since a slow algorithm will bring us back to the long-delay pipeline stage, and fairness, since having many task graphs that have nothing knocking their doors makes them useless. The fairness property must be ensured not only spatially, but also temporally. This can be explained by describing the best and worst case scenarios. Let's say that we have n task graphs ($TG_0 \cdots TG_{n-1}$), and m items to be distributed. The best case is when having a round-robin-like distribution algorithm. In this case, TG_0 does not get a second item at its input buffer before all other $n-1$ task graphs have also received some inputs. This is to ensure that the different task graphs are busy, while the distribution process goes on as shown in Figure 5.3(A). The worst case on the other hand is when the distribution algorithm gives the first m/n items to the first task graphs, the second m/n items to the second task graphs and so on. This scenario implies that the task graphs are working in a serial fashion as shown in Figure 5.3(B), exactly one after the other, which is equivalent to having one active task graph at a time, in addition to the extra overhead of running the distribution algorithm. Notice that both scenarios have distributed exactly m/n items per task graph eventually.

Given that the data to be distributed are 48-bit memory addresses, and the number of task graphs to choose from is relatively small (5 bits are needed to address 32 task graphs), this problem sounds similar to error detecting codes problems. In our case we should compute the target task graph index as fast as possible (in 1 cycle if possible), therefore, multiple-rounds algorithms, or those which use complex operations such as division should be avoided. Furthermore, since our input data are memory addresses, we noticed that for a certain application, the memory addresses it touches differ only in the lower 20 bits.

For those reasons, we empirically used the following algorithm to compute the target task graph index:

$$\begin{aligned} TaskGraphID = [&addr(19..15) \oplus addr(14..10) \\ &\oplus \ addr(09..05) \oplus addr(04..0)] \\ &\text{mod} \ \ num\_task\_graphs; \end{aligned}$$

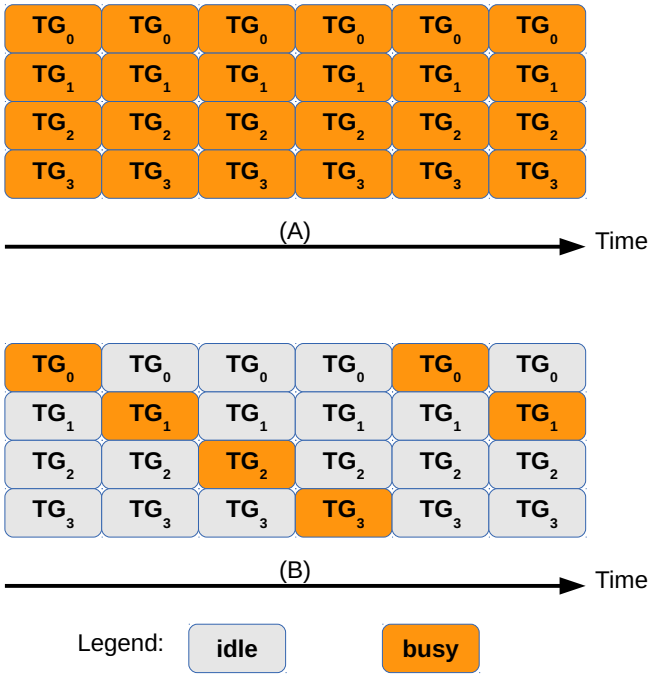


Figure 5.3: (A) Best vs. (B) worst case scenarios of the utilization of 4 task graphs.

The proposed algorithm is based on simple XOR operations of the lowest 20 bits of the input address, divided into 5-bit blocks. It can be computed in one cycle, and has shown experimentally good distribution of the input data among the task graphs as shown in Figure 5.4, regardless of the used number of task graphs, up to 32 though.

After having distributed all the memory addresses in the new task's input/output list, the *Input Parser* stores the new task in the *Task Pool*. This is important at the end of a task's life cycle; i.e., after running it. At this point, the runtime system should report the task as a finished task, and the *Input Parser* will read its input/output list from the *Task Pool*, and read their entries in the different task graphs using the same algorithm, in order to update the task graphs subsequently.

5.2.3 Data Insertion into Task Graphs

The insertion process starts at each task graph whenever it receives data from the *Input Parser*. Additional buffers must be added before each task graph, namely the *New Args. Buffers*, in order to decouple the *Input Parsing* and *Insertion* processes.

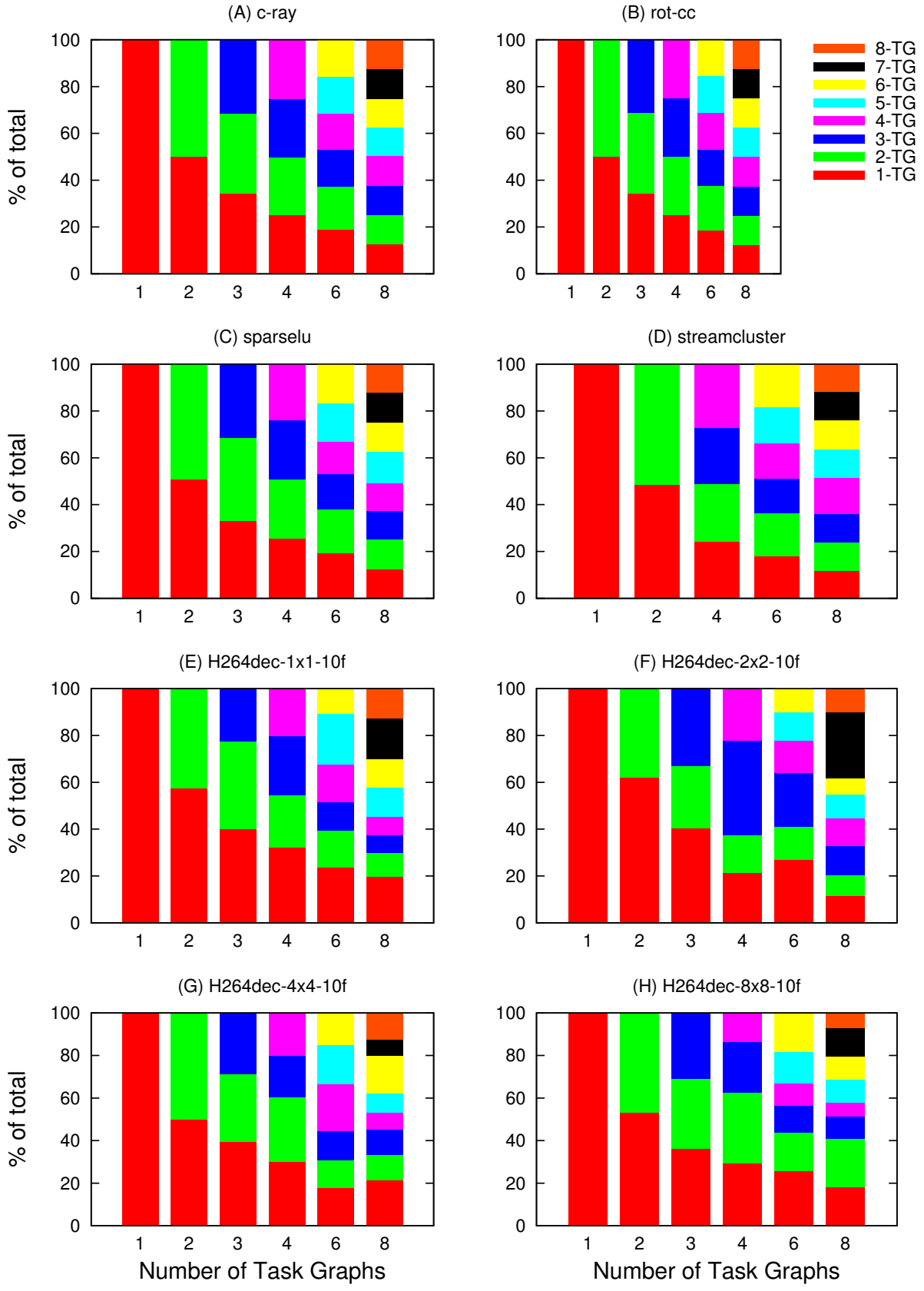


Figure 5.4: The distribution of the input memory addresses of the set of benchmarks (described in Section 5.3.1) among variable number of task graphs.

The same principle is applied in case that the incoming task is a finished task, in which case the *Input Parser* will read its input/output list from the *Task Pool*, and distribute them subsequently among the *Finished Args. Buffers* shown in Figure 5.2.

Each one of the task graphs shown in Figure 5.2 is the same as the one used in Nexus++ described in Chapter 4. It uses the same set-associative data structure to maintain a *Kick-Off List* for each incoming memory address.

When processing the data in the *New Args. Buffers*, one of two scenarios might occur. The first one is when the task has only one parameter that is to be inserted for the first time in the task graph. This means that the processed task has no other parameters at other task graphs, and therefore can be immediately reported as a ready task. This helps to shrink the size of the last *gather* step in our *scatter-gather* approach. This kind of ready tasks are written at the *Rdy Tasks Buffer* shown below every task graph in Figure 5.2.

The second scenario, is when the new task found dependent, or when it has other parameters to be inserted in other task graphs. In this case, the result is written in the *Dep. Counts Buffer* shown below every task graph in Figure 5.2, indicating the task's id, and its dependence count: how many *Kick-Off Lists* has it been added to in that task graph only.

The *gather* step then takes place by the *Dependence Counts Arbiter* whenever any of the *Rdy Tasks Buffer* or the *Dep. Counts Buffer* is written. If any task was reported as ready, the *gather* step in this case will be an arbitration of writing them to the *Internal Ready Tasks Buffer*, in order to be forwarded to the *Nexus IO* unit.

When gathering the results from the *Dep. Counts Buffers* on the other hand, the *gather* step is relatively longer. It should collect the results from the different task graphs, and conclude the final dependence count of each incoming task.

Having a distributed approach, some parameters of a certain task might be processed by other task graphs sooner than the others because of many factors. For example if one task graph stalled and the other not, or if they stalled for different periods of time. It can also be because one task graph received more data to process than the other. In such cases, while waiting for all the parameters of a certain task to be processed by the different task graphs, the temporal dependence count of this task is stored at *Sim(-ultaneous) Tasks Dep. Counts Buffer* shown in Figure 5.2.

Having all task's parameters processed and was found ready, the task's ID will be written on the *Internal Ready Tasks Buffer*. Otherwise, its dependence count will be stored in the global *Dep. Counts Table* shown in Figure 5.2 on page 119.

Finally, when processing finished tasks, if there were some tasks waiting in the *Kick-Off List* of a finished task, those waiting tasks will be written in the *Wait. Tasks Buffer* shown below every task graph in Figure 5.2. The *Dependence Counts Arbiter* after that decrements the dependence counts of those waiting tasks one by one, and decides accordingly whether they are ready to run, or not yet.

In the next section we will describe Nexus#'s pipeline, and how it improves over its predecessor.

5.2.4 Nexus# Processing Pipeline

To demonstrate how Nexus#'s pipeline improves over that of Nexus++, we show in Figure 5.5 the pipeline of inserting tasks that have 4 parameters each, which is the same example used to explain Nexus++ pipeline.

The pipeline of Nexus# has four stages. *I*nput *P*arsing, data *IN*sertion to the task graph, dependence counts *AR*biteration as described in Section 5.2.3, and finally, the *W*rite ready tasks *B*ack to the *Nexus IO* unit.

The input parsing stage consumes two cycles to receive the header word of the new task (its function pointer and number of parameters), and another two cycles for each parameter (IP_h and IP stages respectively in Figure 5.5). The communication scheme is based on the PCIe bus used in the FPGA board prototype of Nexus++ in Chapter 4, therefore, one parameter (48-bit memory address) takes two 32-bit PCIe packets, and thus two cycles.

The *Input Parser* directly distributes every incoming parameter to one of the different task graphs, and in our example, after having distributed the four parameters of the new task, this task's descriptor gets written to the *Task Pool* in one cycle (IP_f stage in Figure 5.5).

$fifo_{1-4}$ in the pipeline are the *New Args. Buffers* described in Section 5.2.3. The date written to them needs 3 cycles to appear at their output, which will trigger the next stage in the pipeline: Data Insertion into the task graph. The latter stage takes

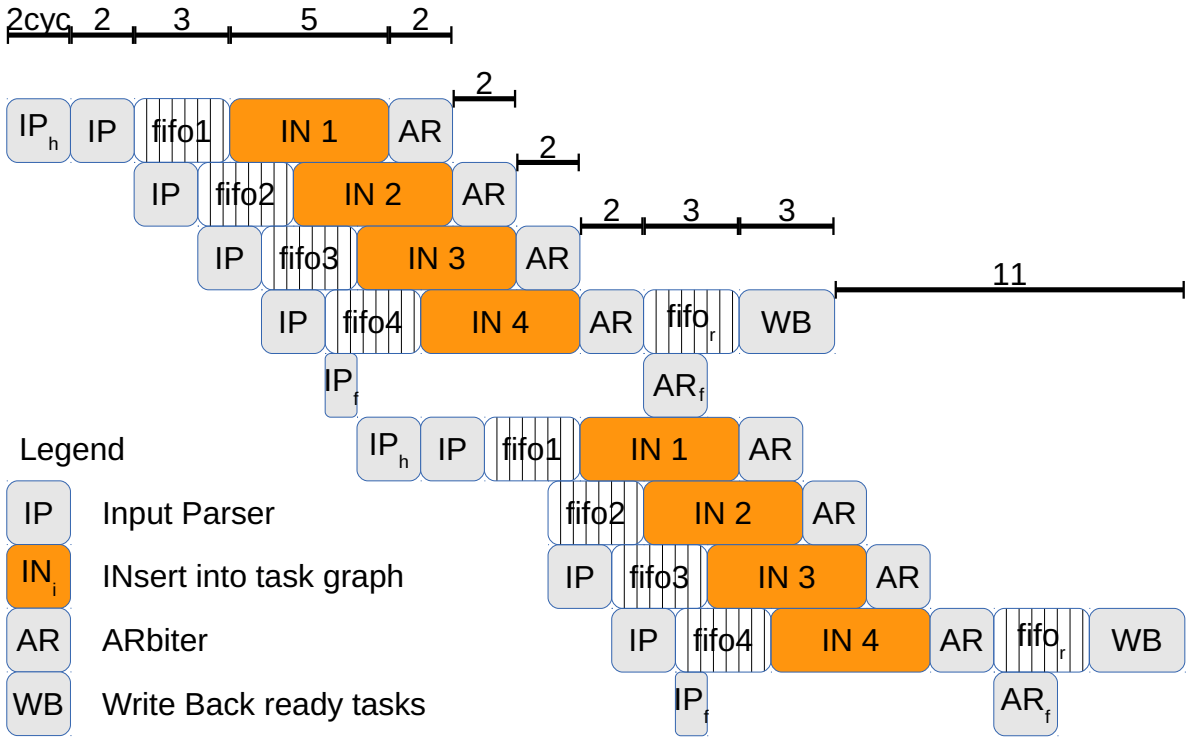


Figure 5.5: Nexus# average-case pipeline.

5 cycles per parameter, and once done, the *Dependence Counts Arbiter* collects its result(AR stage in Figure 5.5).

Once having collected the results of all the 4 parameters of the inserted task in our example, the arbiter checks the readiness of the task. If the task was found ready, its ID will be written to the *Internal Ready Tasks Buffer* ($fifo_r$ in Figure 5.5). The latter fifo takes also 3 cycles to get its data readable at its output port, after which the last stage of the pipeline (*Write Back*) takes place. The latter stage consumes 3 cycles, and simply reads the actual function pointer of the ready task from the *Function Pointers* table shown in Figure 5.2, and forwards it to the *Nexus IO* unit, to be read later by the runtime system.

The difference between the two pipelines can be obviously seen comparing Figures 5.1 and 5.5. The insertion stage in the new pipeline consumed 11 cycles, compared to 18 cycles in the old pipeline. Furthermore, the insertion stage does not wait for the all the task's parameters to arrive in order to start inserting the first one. Also interesting, the write back stage, where ready tasks are forwarded to the *Nexus IO* unit, took place every other 18 cycles in the old pipeline for our example. This number decreased significantly to 11 cycles in the new pipeline.

Although the pipeline shown in Figure 5.5 is an example for inserting tasks of 4 parameters each, in real runs the pipeline will be different. The pipeline shown in Figure 5.5 assumes that the input buffers of the task graphs are empty, hence the pipelined parallel insertion of the different parameters. If on the other hand the 4 parameters of our example task were already in the buffers, the pipeline in this best-case scenario will behave as shown in Figure 5.6. In this scenario, the *Write Back* stage will take place every other 5 cycles. It is worth mentioning that in this case, the arbiter consumes only two cycles to collect the results of all the task graphs and conclude the final dependence count of the corresponding tasks.

There are also scenarios where the insertion stage takes longer periods of time, if for example the task graph stalled due to not fining an empty slot for a certain line in the set-associative structure. The task graph must then wait until one task finishes, which its parameters share the same line. The good thing about such a scenario is that this gives time to the *Input Parsing* stage to fill up the input buffer of the stalled task graph, increasing the chance that all task graphs work in parallel as in the best-case scenario shown in Figure 5.6.

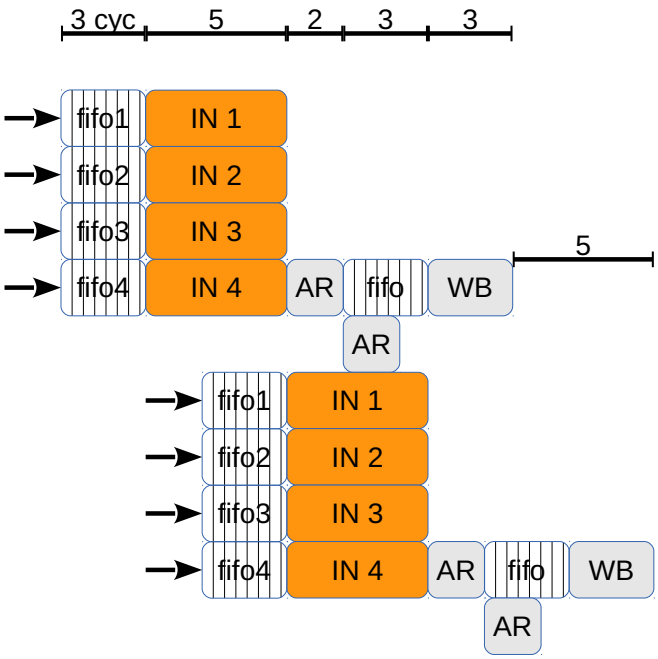


Figure 5.6: Nexus# best-case pipeline.

The *Dependence Count Arbiter* handles a relatively large amount of computation, which might eventually make it a bottleneck. To avoid this, we designed it in a way to iterate between the three buffers at the end of each task graph in a prioritized fashion. The highest priority goes to reading the *Ready Tasks Buffer*, since they are ready tasks and only need to be forwarded to the next pipeline stage. Second priority is for reading the *Waiting Tasks Buffers*, since they have potential ready tasks. While serving one of the previous two scenarios, this gives time for the different task graphs to finish what they do, namely inserting new items to the task graph. Therefore, this increases the chance of reaching the best-case scenario pipeline shown in Figure 5.6. To accomplish this, the lowest priority in the *Dependence Count Arbiter* is for reading the *Dep. Counts Buffers*.

In the whole design process, we made sure that Nexus# is deadlock-free, by well-dividing it into different blocks, with fifo lists used as the communication medium to ensure decoupling, and testing it thoroughly.

Configuration	Registers	LUTs	Block RAMs	Maximum (Test) Frequency(MHz)	Total Utilization
ZC 706 (Totals)	437200	218600	545		
Nexus++	1%	7%	14%	114.44 (100.00)	7%
Nexus# 1 TG	1%	8%	13%	112.63 (100.00)	7%
Nexus# 2 TGs	2%	15%	25%	112.63 (100.00)	15%
Nexus# 4 TGs	3%	29%	47%	85.26 (83.33)	29%
Nexus# 6 TGs	4%	44%	69%	55.66 (55.56)	44%
Nexus# 8 TGs	4%	58%	91%	43.53 (41.66)	58%

Table 5.1: Device utilization using different design configurations on the ZC706 FPGA board.

5.2.5 Nexus# Synthesis

Nexus# synthesizability was tested on the Xilinx ZYNQ-7 ZC706 FPGA board. This board has a relatively larger FPGA compared to the Virtex-5 board used in the evaluation of Nexus++ in Chapter 4. In Nexus#, we wanted to evaluate the distributed task graphs paradigm, which did not fit on the Virtex-5 FPGA board, and thus the switch to the ZYNQ-7 ZC706 FPGA board. Table 5.1 shows an overview of the target FPGA utilization, using different design configurations.

Our baseline is the Nexus++ design in Chapter 4. Although it was evaluated using a different FPGA board, we have re-synthesized it using the ZC706 FPGA board to make it comparable with the other configurations.

The three main criteria shown in the table are the registers, look-up tables(LUTs), and block RAMs. The latter reflects the data structures used in the design, mainly the tables in the task graphs for example, while the first two reflect the computational part of the design, i.e., the state machines.

Having only one task graph in the configuration of Nexus# is most analogous to Neuxs++. This can be seen in Table 5.1, as both have very close utilization values.

By increasing the number of task graphs in Nexus#, it can be noticed how this is reflected in Table 5.1: the number of block RAMs almost doubles due to using multiple task graphs, and the number of LUTs also doubles because of the extra work the *Input Parser* and the *Dependence Counts Arbiter* blocks have to manage every time the number of task graphs doubles.

	# tasks	total work (ms)	avg task size (μs)	avg #deps	total #deps
c-ray	1200	7381	6151	1	1200
rot-cc	16262	8150	501	1	16262
sparselu	54814	38128	696	1-3	155452
streamcluster	652776	237908	364	1-3	653168
h264dec-1x1-10f	139961	640	4.6	2-6	760308
h264dec-2x2-10f	35921	550	15.3	2-6	192695
h264dec-4x4-10f	9333	519	55.6	2-6	49045
h264dec-8x8-10f	2686	510	189.9	2-6	13481

Table 5.2: Benchmarks' Durations obtained from traces collected on Xeon E7-4870.

5.3 Performance Evaluation

To evaluate the performance of Nexus#, we performed several trace-based simulations using various benchmarks. The purpose of our experiments is to measure the scalability enhancements that can be obtained by the new distributed task graphs.

5.3.1 Benchmarks

The set of benchmarks used to evaluate Nexus# are from the Starbench benchmark suite [6], which are described in Chapter 4 and were used to evaluate Nexus++. These include *c-ray* (ray tracing), *h264dec* (H.264 video decoding), *rot-cc* (image rotation and color conversion) and *streamcluster* (k-median clustering). To these benchmarks, we add *sparselu* (sparse LU matrix factorization benchmark used by the OmpSs developers).

Moreover, the task size in *h264dec* was increased by processing several (8×8) macroblocks in one task as the optimum tradeoff between paralellism and runtime overhead for the Nanos runtime [6]. In the evaluation of Nexus#, we vary the task size for the *h264dec* to check the effect of this on the scalability of Nexus#, especially that the latter supports the *taskwait on* pragma. Table 5.2 lists the main characteristics of the set of used benchmarks.

We test 4 variation of the *h264dec* benchmark varying the number of macroblocks that are mapped to one task. *h264dec-1x1-10f* indicates that only 1 macroblock is mapped to one task, *h264dec-2x2-10f* mappes 4 macroblocks to one task, and so on. All variation has 10 full HD frames (hence the 10f) of a video stream (pedestrian_area.h264) as input.

To validate the dummy tasks/entries approach, the task graph of Gaussian elimination with partial pivoting is used, which is presented in Chpater 3. In this benchmark, the number of tasks that depend on a certain memory segment can grow very large since it depends on the size of the input matrix, as depicted in the dependency pattern of Figure 3.8, assuming an $n \times n$ matrix.

5.3.2 Experimental Setup

From the execution of each benchmark on a 40-core Xeon E7-4870 machine running at 2.40GHz, we collected traces that include the task descriptors (which specify the inter-task dependencies) and the execution time of each task. The test bench simulates the runtime system. It submits new tasks to Neuxs#, receives ready task information from it, schedules ready tasks to worker cores and simulates their execution, and finally notifies Nexus# of finished tasks. Using the information from the traces, the same set of simulations presented in Chapter 4, Section 4.3.3, were performed, namely:

No Overhead: This simulates the execution of an application without any overhead, to determine the lower bound for the execution time of the benchmarks. In this simulation, the simulation time does not advance while dependencies are resolved. Only the execution time of the tasks is taken into account. This allows us to determine when the lack of available parallelism in the application is the limiting factor.

Nexus# only: In this simulation, we additionally account for the dependency resolution overhead incurred by the Nexus# core. If an application scales much worse in this simulation than in the *No Overhead* experiment, this indicates a bottleneck inside the Nexus# design. In this simulation free worker cores start executing tasks directly after they are reported as ready by Nexus#. No communication or other non-dependency resolution overhead is accounted for.

Nexus# and runtime: Here, an additional delay of $(n+1)*250+400$ns is introduced between Nexus# reporting a task as ready and the start of execution of the task by the worker core, as well as a delay of 250ns between the end of the task and the reception of the finished task notification by Nexus#. This represents the overhead of communication between the processor cores and Nexus#, as described in Section 4.3.2 Additionally, Nexus# cannot replace other runtime duties such as setting up the stack for the task and switching execution to it. We measured the overhead of these features

to be approximately 5μs per task. In this simulation, task length is increased by this constant, to account for all necessary parts of execution.

These simulations are compared to the actual runs of the benchmarks on the same machine that the traces were collected on, compiled using the Mercurium compiler version 1.3.5.8 and linked to the accompanying Nanos runtime library. We also compare them to the results obtained when using Nexus++ as the task manager.

5.4 Evaluation Results

5.4.1 Design Space Exploration

The first experiments we carried out to evaluate Nexus# were to explore the design space: we simulated the H264dec benchmarks with changing the number of task graphs (TGs) used in Nexus#, in order to get the optimal configuration. We chose the H264dec benchmark because we can group several macroblocks to be decoded by one task, and hence varying the task size. The finer the tasks are, the more challenging it becomes for any task graph manager, since the worker cores will finish executing their fine tasks quickly and demand the scheduler for new tasks more often. This way, we can see the impact of the task size on the performance Nexus#.

Grouping several macroblocks together is not a trivial task, and requires the programmers to explicitly specify which macroblocks can be grouped together in order to preserve dependencies. The goal of Nexus# is to alleviate the programmer from doing this, by being able to manage the most fine-grain tasks without the need to apply the grouping technique (1 macroblock per task), as discussed in Section 2.2.

The results of the different scalability tests are depicted in Figure 5.7. The graphs in Figure 5.7 show the results of running Nexus# at 100MHz, regardless the number of task graphs used. This is to give a fair scalability test and relating it to only the number of task graphs used. Figure 5.8 on the other hand, depicts the results of running Nexus# at variable frequencies depending on the number of task graphs, as per Table 5.1. This gives the realistic performance of Nexus# using a certain number of task graphs, and helps to determine the optimal configuration of Nexus#. Furthermore, Figure 5.8 shows the results of a second set of experiments on the right-hand side, adding runtime and communication overheads to the simulations. This is to show the effect of the runtime overhead on the scalability behavior, when dealing with tasks having different granularities. All speedup results are calculated against the single core execution time of the ideal curve.

No grouping of macroblocks exist in Figures 5.7(A), 5.8(A, A'), meaning that these experiments have the most fine-grain tasks. The other graphs include grouping of macroblocks: from top-to-bottom order of both figures, 2×2, 4×4, and 8×8 macroblocks per task respectively. Looking back at Table 5.2, one can see the effect of macroblock grouping on task size, which went from 4.6 μs on average in case of no

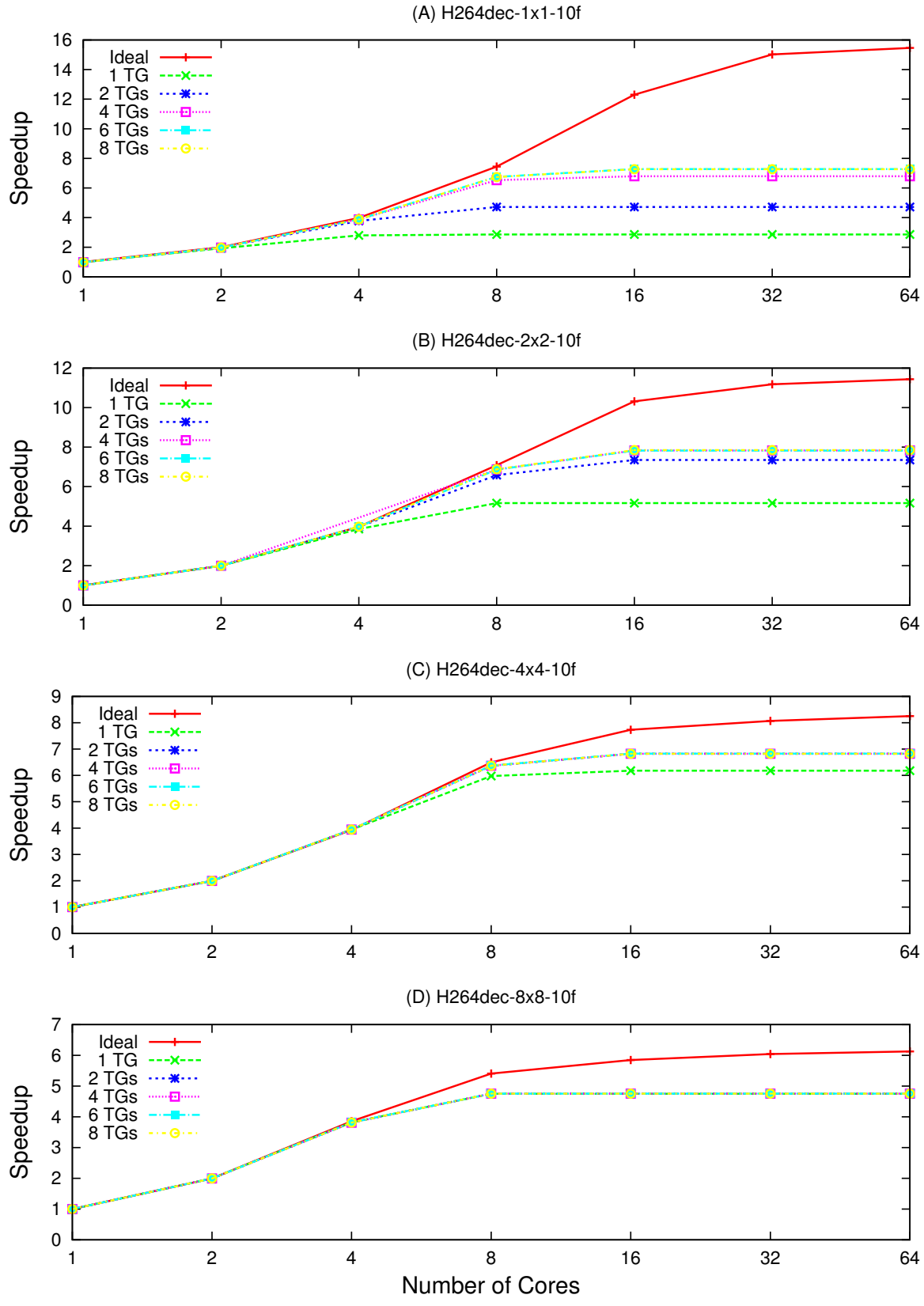


Figure 5.7: Scalability of Nexus# running different configurations of the H264dec benchmark, at 100 MHz.

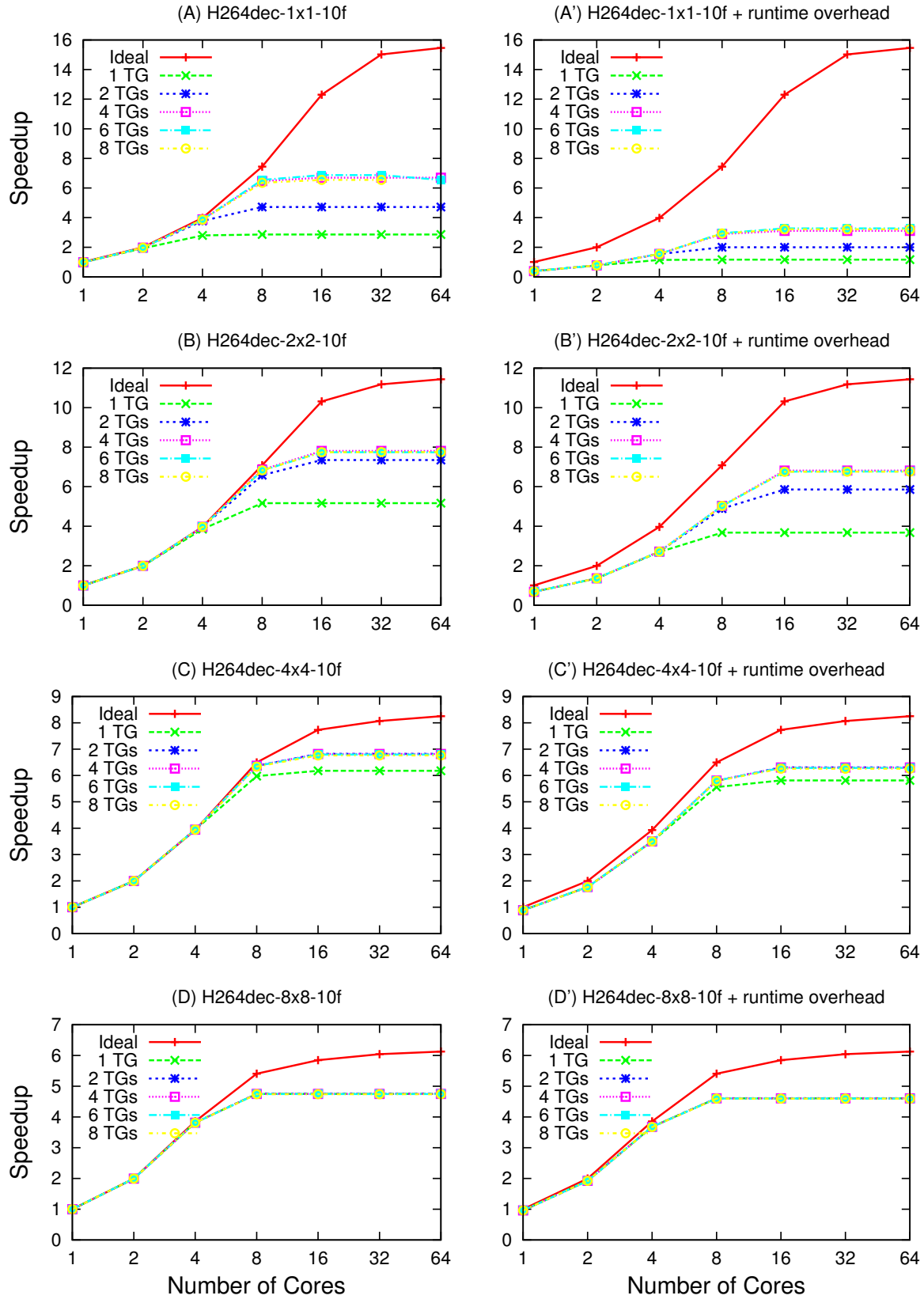


Figure 5.8: Scalability of Nexus# running different configurations of the H264dec benchmark, running at a variable frequency shown in Table 5.1. Right-hand side figures also depict the runtime overhead.

grouping, to about 190 μs in case of grouping 8×8 macroblocks per task. Having the same input, the total number of tasks also changed drastically between the different configurations.

The red line (most-upper in all graphs) in Figure 5.7 is the ideal scalability curve, where only the execution times of tasks have been simulated, without adding the time needed to resolve dependencies or other runtime overhead as described in Section 5.3.2, the *No Overhead* experiment.

It can be seen that the larger the task size is, the easier it becomes for Nexus# to handle tasks, since it can get closer to the ideal scalability curve, even using small number of task graphs. Most interestingly is the hardest experiment shown in the upper-most graphs in Figures 5.7, 5.8, where Nexus# scales up to 7×, using 6 task graphs.

The differences between using 4, 6, and 8 task graphs are very minimal, but we chose to use 6 task graphs for our later evaluation, since this configuration achieves the best scalability results, and that best utilizes the look-up tables (LUT) in the target FPGA. Basically, the LUTs on our target FPGA board support six-input and two-output logic, which means that a 6-input adder, for example, can be mapped to one LUT, which is one main instant used by the *Dependence Counts Arbiter* described earlier in Section 5.2.3.

For the case of the target FPGA, implementing a logic circuit that has more than 6 inputs, an 8-input adder for example used by the 8-task graphs configuration of Nexus#, means that cascading of two LUTs is necessary to realize this logic circuit on the FPGA. Cascading LUTs affects the maximum clock frequency at which the final design can operate, as shown in Table 5.1 when increasing the number of used task graphs.

Furthermore, changing the number of task graphs in Nexus# impacts the maximum operating frequency as shown in Table 5.1, since the design has more structures as the number of task graphs grows. This imposes more work on the *Dependence Counts Arbiter*; the unit responsible for gathering results from the different task graphs.

Hardware-wise comparison with a major related work by Yazdanpanah et al. [173, 174] shows that their design consumes 29,138 registers and 110,729 LUTs respectively, which is comparable to the resources needed by our 8 task graphs design

(19,350/127,290 registers/LUTs respectively), and 6× more than the resources needed by the 1 task graph configuration. Moreover, using a micro benchmark built after [173] that includes inserting 5 independent tasks, each with two parameters, Nexus# (with one task graph) consumes 78 cycles compared to 172 cycles consumed in [173]. Their design can run at a higher frequency though.

The results of the two experiments shown in Figure 5.7, with Nexus# running at 100MHz and in Figure 5.8, with it running at the test frequency shown in Table 5.1, they both confirm our observation that using 6 task graphs achieves the best scalability results. Although the operating frequency has been reduced significantly in the experiments of Figure 5.8 using 6 and 8 task graphs to 55.56 MHz and 41.66 MHz respectively, their performance results were slightly smaller than their higher speed counterparts in Figure 5.7.

The runtime overhead effect shown in the right-hand experiments of Figure 5.8 differs significantly as the granularity of the tasks change. In the case of very fine-grain tasks, the runtime overhead reduces the scalability significantly, since this slows down the incoming tasks stream to Nexus#, as well as the finished tasks notifications. This effect gets slowly less significant as the granularity of tasks increases as shown in Figure 5.8(B', C', D') respectively.

5.4.2 Benchmarks Scalability

Based on the design space exploration experiments presented in the previous section, the following experiments were performed using 6 task graphs running at 55.56 MHz. First, we evaluate the benchmarks listed in Table 5.2.

Figures 5.9 and 5.10 show the performance evaluation of Nexus# using 6 task graphs, and compares it to other task graph managers: 1- the OmpSs runtime system called Nanos, and 2- Nexus++. Furthermore, the ideal scalability curve for each benchmark is also added to the different graphs in order to see the big picture, i.e. how close are the different task managers to achieving the ideal scalability.

All speedup results are calculated against the single core execution time of the ideal curve, which is very close (although faster) to the sequential version of the benchmark. The results of Nanos are only up to 32 cores, which is limited by the hard number of cores our test Xeon E7-4870 machine has. Figure 5.9 left-hand side shows the

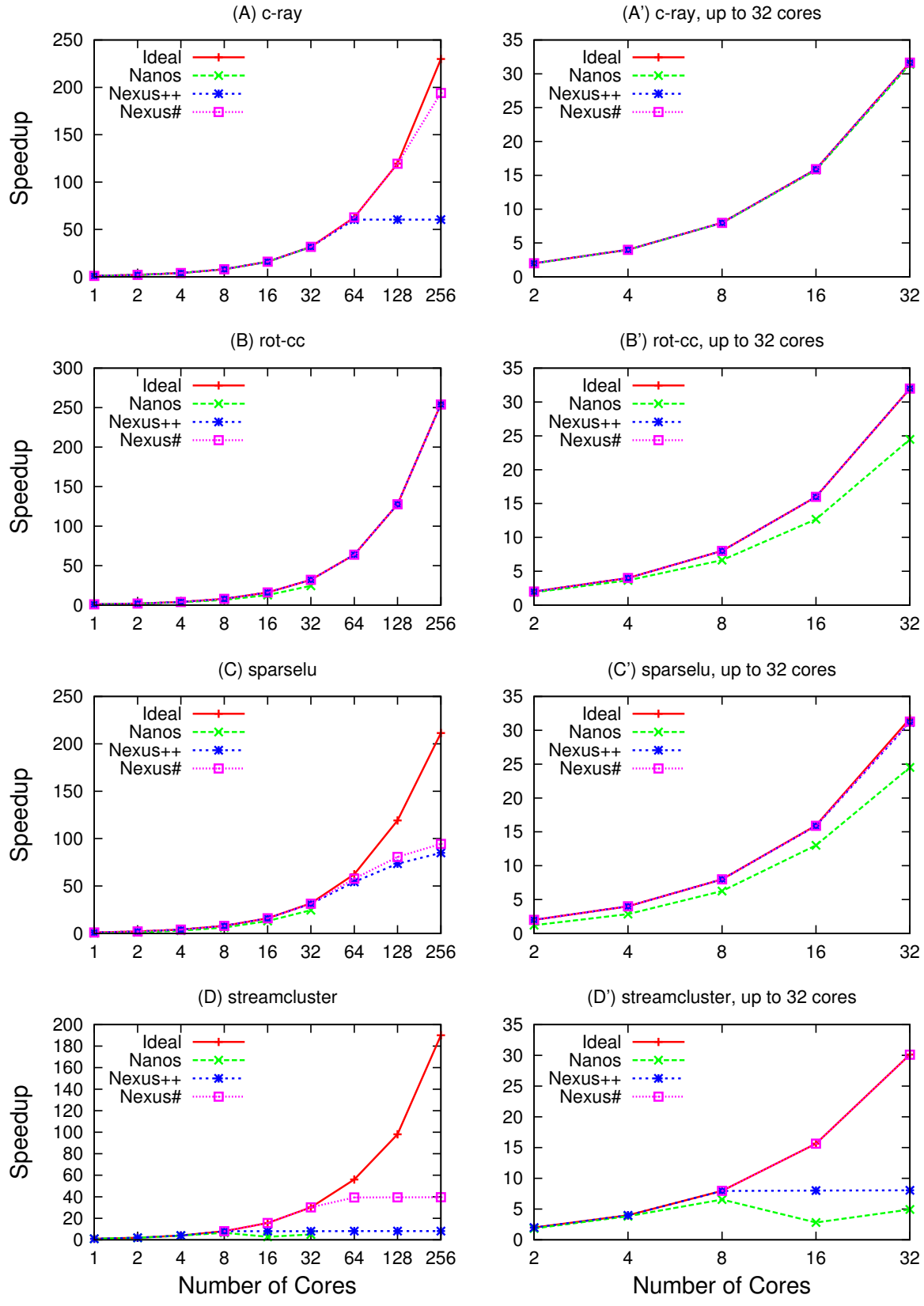


Figure 5.9: Performance of Nexus# running different benchmarks, in comparison to other task managers.

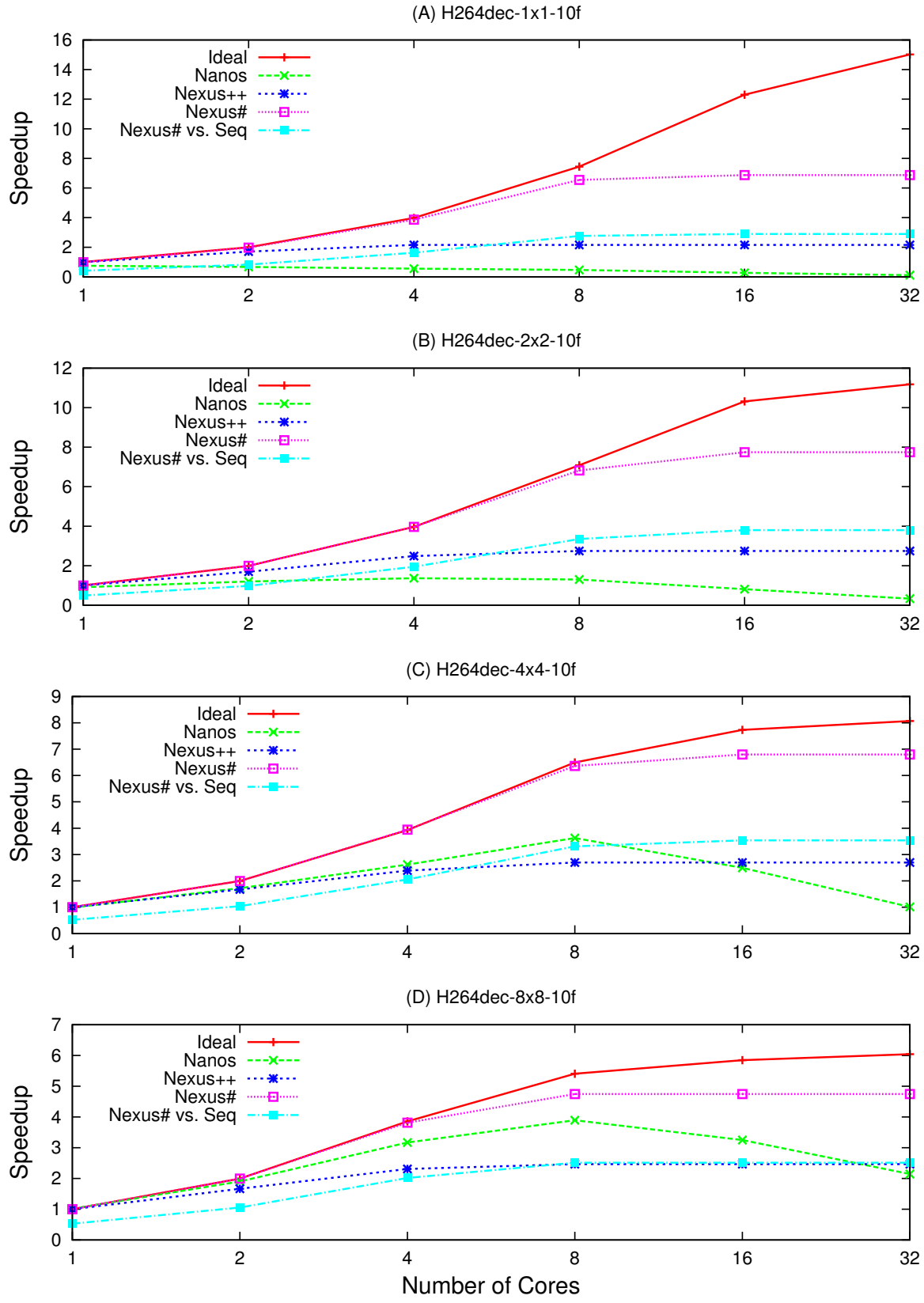


Figure 5.10: Performance of Nexus# running different configurations of the H264dec benchmark, in comparison to other task managers.

scalability of the different benchmarks varying the number of cores from 1 up to 256. As this limits the clarity of the results for cores up to 32, the results for cores up to 32 are shown separately on the right-hand side of Figure 5.9.

In Figure 5.9(A), the *c-ray* benchmark represents an easy case for all the task managers; it has relatively large tasks (6 msec on average), and has only independent tasks. All task graph managers performed well and were close to the ideal scalability curve and scored 31.5× speedup on 32 cores. Nexus# continued scaling and scored 194× speedup on 256 cores, compared to 60.4× achieved by Nexus++.

The *rot-cc* benchmark has smaller tasks, with pairwise independent tasks, which is a harder case for the task graph managers than *c-ray*, but still relatively easy. Both the hardware task managers (Nexus++ and Nexus#) scored 32× speedup on 32 cores, which is better than the software task manager (Nanos) which scored only 24×. Nexus# and Nexus++ continued scaling to 256 cores, both achieving 254× speedup.

The *sparselu* benchmark has more complex dependencies between its tasks, and again, the hardware task managers performed better than Nanos (31× vs. 24.5× speedup on 32 cores respectively). Nexus# achieved up to 94.4× speedup on 256 cores, which is slightly better than the 84.9× speedup achieved by Nexus++.

The *streamcluster* benchmark is a more difficult one for the task graphs. Nanos achieved only 4.9× on 32 cores, whereas Nexus++ and Nexus# achieved 7.9× and 30.1× respectively. Nexus# continued scaling achieving about 39× speedup on 64 cores.

Figure 5.10 shows the scalability results of the H264dec benchmark for up to 32 cores, since this benchmark, in its different configurations, did not scale beyond this number of cores. Grouping several macroblocks per task increases the average task size, and makes the management of the task graph much easier. Nanos in particular achieved its best performance when 8×8 macroblocks are grouped in one task, and scored 3.9× on 8 cores. Its performance dropped down when using larger number of cores though. Nexus# achieved slightly better speedup, and sustained its performance for larger number of cores. Our main focus is on Figure 5.10(A), the experiment in which the programmer does not do any grouping of macroblocks per task. In this figure, we can see that Nanos did not achieve any speedup. Nexus# on the other hand achieved up to 6.9× on 16 cores. Nexus++ does not support the "task-wait-on" OmpSs pragma, and achieved only 2.2× speedup on 4 cores.

Benchmark	Nanos Max.	Nexus++ Max.	Nexus# Max.
c-ray	31.4×	60.4×	194.0×
rot-cc	24.5×	254×	254×
sparselu	24.5×	84.9×	94.4×
streamcluster	4.9×	7.9×	39.6×
h264dec-1x1-10f	0.7×	2.2×	6.9×
h264dec-2x2-10f	1.4×	2.7×	7.7×
h264dec-4x4-10f	3.6×	2.7×	6.8×
h264dec-8x8-10f	3.9×	2.5×	4.7×

Table 5.3: Maximum achievable speedup using the different task graph managers.

From all the graphs in Figures 5.9 and 5.10, it can be clearly seen that Nexus# has the upper hand over the other two task graph managers. Most interestingly are the cases where tasks are very fine grained. Such tasks are the key to utilizing the ever-increasing computing power embedded on the state-of-the-art and future SoCs. In such cases, the strength of Nexus# is mostly clear and needed. Table 5.3 summarizes the maximum achievable speedup for the different benchmarks using Nanos, Nexus++, and Nexus#.

Since we based our evaluation results on the single-core ideal simulations, it is also worth mentioning that the sequential version for most of the benchmarks has a very close (slower) execution time as our baseline. The only exception is the H264dec benchmark, where the sequential execution is almost twice as fast as the single-core ideal simulation, resulting in a real speedup of Nexus# versus the sequential execution time half of that versus the single-core ideal simulations as can be shown in Figure 5.10. This shows the potential overhead of porting an application to the dataflow task execution model. Hence, the maximum (real) speedup achieved by Nexus# in the case of H264dec benchmarks is about 3×.

Figure 5.11 shows the speedup achieved when running the Gaussian elimination problem (Figure 3.8 on page 76) on different multicore systems for different matrices of sizes ranging from 250×250 to 3000×3000. The Gaussian elimination benchmark is a micro benchmark that is not trace-based as the previous benchmarks. This benchmark in particular is a worst-case scenario for Nexus# as the example described in Figure 5.3(B) on page 121.

Running the application on a 250 × 250 matrix for example, starts by having one ready task (T_1), and 249 dependent tasks. Those are direct dependencies, meaning that all the 249 tasks have the same memory address as input, that is to be produced by the first ready task (T_1). This indicates that regardless the number of task graphs used in Nexus#, only one will be used to insert the first 250 tasks, and another one for the next wave of tasks, and so on. For this reason, increasing the number of task graphs used for this benchmark will have a negative impact on performance, mainly because the clock frequency deriving Nexus# decreases as the number of task graphs increases. Furthermore, the tasks generated in this benchmark has only up to 2 parameters per task. As a result, we chose to evaluate this benchmark using only up to 2 task graphs.

Figure 5.11 shows the results of the Gaussian elimination benchmark using Nexus++, Nexus# with one task graph (1TG), Nexus# with two task graphs (2TG) respectively, all running at 100 MHz. The baseline here is the single-core execution time using Nexus++. Each worker core is assumed to be able to do 2 GFLOPS, which means that the average computation time (in μs) of the Gaussian tasks = 2000/#FLOPs, as shown in Table 3.3 on page 77 in Chapter 3. It can be seen that Nexus# (2TG) has a slight improvement over Nexus++. About 19% in case of the very fine grain tasks in matrix-250. As the matrix size increases (and hence larger number of tasks of larger granularity), Nexus# has about 10% performance improvement over Nexus++. This benchmark is to show that our hardware task managers do not have a static limit on the number of tasks that can wait for a certain memory address.

According to Vandierendonck et al. [160], the runtime overhead of their proposed software task graph manager can go as low as 400 cycles (0.2 μs on their test machine) per task, their experiment assumed inserting 1-parameter tasks to an empty task graph, which is an ideal case. Therefore, the hardware acceleration is vital.

Designing a hardware accelerator for the runtime system in a multicore system implies that a communication between the two to take place in real time. Our hardware task manager is meant to be integrated in future many-core chips, which in the era of high integrity and homogeneity. We believe that a FPGA will also be integrated on the same chip. This will provide a low-latency communication channel between the computing cores and the user design on the FPGA, which opens the door to more research and development. Eventually, Nexus# as in the case of other hardware

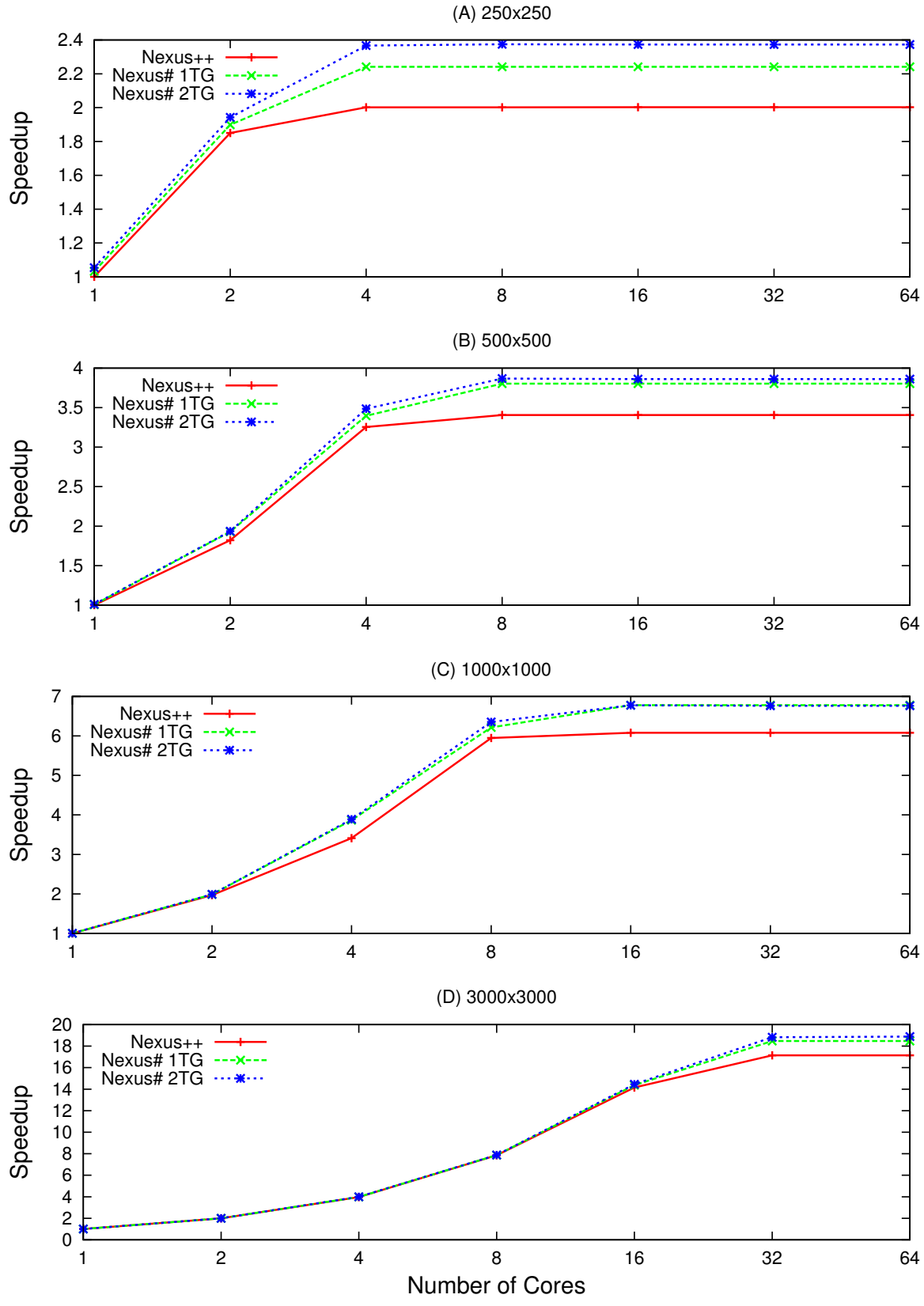


Figure 5.11: Performance of Nexus# running Gaussian elimination benchmark for different matrix sizes.

co-processor, would achieve its best performance when implemented as an ASIC co-processor.

Compared to the state-of-the-art clock frequencies that drive microprocessors found in HPC machines or even consumer products, Nexus# runs at a relatively very low frequency (50 - 100 MHz). Although power-consumption analysis is part of the future work, Nexus# has a potential to manage the task graphs in a wide range of multicore machines, without becoming the power-drain hot spot. Depending on the use case, the needed number of task graphs can be accordingly configured. In cases with limited number of cores and/or limited power, as in smartphones and other consumer devices, it can even be turned off (as dark silicon) if the number of ready tasks exceeds a certain threshold.

The door is wide open ahead of Nexus# to be integrated in real multicore/manycore SoCs, as we think that the task-based dataflow execution model is a key to utilizing the computational power of such systems.

5.5 Summary

This chapter presents Nexus#, a VHDL prototype of a hardware task manager for the OmpSs runtime system. Supporting the in, out, inout, taskwait, and taskwait on pragmas, Nexus# is suitable for wide range of applications, including H264 decoding. It employs a distributed task graph management strategy, which enables the parallel insertion of the input/output memory addresses of the incoming tasks. Besides implementing a low-latency task graph look-up mechanism using set-associative cache-like structures, Nexus# also uses a fast distribution function that efficiently directs the incoming memory addresses to the proper task graph.

Generating data and runtime traces for multiple benchmarks in the Starbench suite, and embedding them in a Modelsim testbench, the experimental results show that Nexus# achieves significant speedups for all the benchmarks on a 256-core pseudo-machine. Results also demonstrate that Nexus# performs better than Nanos, the official OmpSs runtime, as well as Nexus++, our central task graph manager, by orders of magnitude, for benchmarks that have very fine grain tasks and/or complex dependency patterns.

It is also shown that a benchmark modeled after Gaussian elimination, where the number of tasks that depend on a certain task is not constant, ran successfully and efficiently with a speedup of 19× for a 3000×3000 matrix using 64 cores.

Nexus# is fully configurable, and depending on the use case, the number of task graphs can be changed. Although targeting OmpSs applications, Nexus#'s low-latency retrieval task graphs can be used with other programming models.

6 Future Perspectives

The main goal of this thesis is utilizing fine-grain task parallelism with irregular dependency patterns in a quest to conquer multicore systems without compromising the easiness of programmability. This would lead to reaching the maximum performance of high-performance computers in general applications, not only by the specially-tuned benchmarks. Furthermore, well-utilizing the multiple cores in an embedded SoC for example, enables delivering the required performance using energy-efficient cores clocked at small frequencies, resulting in lower power consumption and an extended battery life.

Nexus++ and Nexus# present hardware architectures that support runtime systems of the dataflow task-based programming models, focusing on OmpSs. They demonstrated improved scalability of OmpSs applications when the task graph is being managed by either of them. Nexus# has better performance and reconfigurability that can be well tuned to match the target system. In particular since Nexus# has multiple task graphs that process the incoming tasks' information in parallel, the number of task graphs can be varied.

After having implemented the presented both Nexus++ and Nexus# hardware accelerators, this is the right place to answer the question whether this effort is feasible or not? In my opinion, task-based parallel programming models are very appropriate answer to the problem of programmability of multicore, heterogeneous systems. They do not present a challenge to the programmer, yet they enable him/her to utilize the available resources in his/her current and future machine. The runtime overhead is the bottleneck of this approach, and even if in the future a very light weight runtime system has been developed, its thread(s) will be sharing the available resources with the worker threads, limiting the worker threads from utilizing the full potential of the system.

Moreover, adding hardware acceleration for the runtime system that can be used by various applications, is a better idea compared to adding custom hardware for only certain applications. Not only do the proposed hardware task managers reduce the conflict on the available resources as the runtime thread is relieved from managing the data structures of the task graphs, Nexus++ and Nexus# also demonstrated their ability of pushing the scalability of applications that have fine-grain tasks, or complex dependency patterns. So the answer whether Nexus++ and Nexus# designs are feasible or not is absolutely yes.

Possible future perspectives can be categorized as in the following list:

- Improving how Nexus# manages tasks, and improving the handshaking protocol between it and the runtime system, starting with task creation and submission, until the point of task retirement.
- Off-loading more runtime tasks to Nexus#, this requires further analysis of the runtime system and decide on hot spots that can be accelerated by Nexus#. For example, more ready queues can be added to Nexus# where it pushes ready tasks to them based on a pre-determined priority.
- Supporting prefetching, since the memory footprint of the different tasks is recorded in the dataflow graph. Once a task becomes ready to run, the runtime system can be signaled by Nexus# to issue a DMA operation to prefetch the needed data from memory, if they do not already exist in the cache hierarchy.
- Supporting more features in the programming model, for example adding support for nesting tasks, or for virtualization enabling multiple OmpSs applications to run concurrently.
- Supporting more programming models, such as OpenMP 4.0 [23] which, as OmpSs, supports dependency-aware tasking constructs.
- Implementing address renaming techniques in Nexus++ and Nexus#.
- Tightly integrating Nexus++ or Nexus# on chip with modern multicore SoCs. One example was done as a master thesis at TU Berlin [71], where Nexus++ was evaluated as a Tightly coupled co-processor of an ARM-based multicore SoC. The evaluation environment is a Zynq-7000 FPGA board from Xilinx, on which Nexus++ communicates with the multicore system (having two ARM A9 cores) over the AXI bus, and a testbench simulating larger number of cores that do not

exist in the Zynq-7000 SoC. This work can be easily ported to more advanced SoCs, such as Xilinx's Zynq UltraScale+ MPSoC [172] or Altera's Stratix 10 FPGA [4], which both have Quad Cortex-A53 64-bit CPU, in addition to a big FPGA fabric. The simulation results show that Nexus++ scales well for a set of synthetic benchmarks and emphasize on the criticality of the communication interface with the multicore system in order to efficiently integrate Nexus++ with embedded ARM processors, which are found in modern devices such as smartphones and tablets. It also presents a decent experimental environment for improving Nexus++/Nexus# as well as the handshaking protocol with the runtime system.

Other examples of modern multicore architectures include IBM's *Power8* 12-core chip, each core capable of 8 hardware threads simultaneously. This presents a massively multithreaded chip capable of executing total of 96 threads simultaneously. Furthermore, it has a *Coherent Accelerator Processor Interface (CAPI)* [78], which enables connecting custom accelerator to the coherent Power8 SoC. Although meant to bring external accelerators such as GPUs, integrating Nexus++/Nexus# as a co-processor for such a massive multithreaded chip is interesting to bring the most out of OmpSs on the Power8 chip.

7 Conclusions

Nowadays, personal computers as well as battery-powered devices have multiple CPUs, GPU, DSPs, and even FPGA devices forming complex, high performance systems. The heterogeneity also came across CPUs, where some of them are high performance, while the others are power-efficient, as in the case of ARM's big.LITTLE architecture. Demanding applications, such as video processing for example, are assigned to the big cores, while less demanding ones such as audio playback are assigned to the little cores.

The increased complexity of modern multicore systems introduced several challenges to the computing community, including the detection of parallel regions in a program, management of tasks and resources, performance portability across different architectures, application scalability, and programmer's productivity.

This thesis discusses those parallel programming challenges of state-of-the-art and future multicore systems in general, and tackled the bottleneck of dataflow task-based programming models in particular, focusing on making the most out of fine-grain parallelism, while at the same time maintaining easiness of programmability.

Modern systems provide a proper infrastructure for parallel programming for the experienced programmers who are aware of the low-level hardware details of the system. For the average programmer on the other hand, this is not an easy task, since manually dividing a big task into smaller ones and executing them in a proper order according to the dependencies between them is a complex task, especially for complex applications. Furthermore, writing an application manually to run on a certain multicore system is most likely not future-proof, since the programmer is biased by the available multicore system at the time of writing the application. This means that a certain application that can successfully utilize the four cores in a certain system today, might not utilize the eight cores in next year's SoC. Therefore, new programming models emerged addressing those challenges, differing in the degree of relying on the

programmer to do the job of parallel implementation of the application and resource management.

The focus of this thesis is the OmpSs programming model, which abstracts the multicore system to the programmer and reliefs him/her from the burdens of extracting parallelism in the application, by offloading this task to the runtime system. This results in a costly runtime overhead which limits application's scalability.

Although OmpSs and other dependency-aware task-based programming models make it easier for the programmers to utilize a multicore systems, it is beneficial in cases where the resulting tasks out of a certain application are of a big size, since in this case the worker cores can be busy for relatively long time, enough for the runtime system to find the next tasks to be executed, and thus hiding the runtime overhead. Since this is not the case for many applications, the programmer has to interfere in order to take control of tasks' granularity, which brings us to the problem of easiness of programmability and the other parallel programming challenges consequently.

This thesis presents hardware co-processors for the runtime system of task-based programming models such as OmpSs, resulting in reliefing the programmer from the burdens of extracting parallelism, as well as that of controlling tasks' granularity. The co-processor, called Nexus++ in its first prototype, presented a preliminary architecture in the form of a SystemC model, which provided an insight of the optimal configuration of Nexus++, by evaluating the scalability of several micro benchmarks, designed after traces from parallel H.264 video decoding on the Cell processor.

Nexus++ is then implemented as a VHDL module, realized on the FPGA of a Xilinx board with a PCIe bus interface. A wrapper that interfaces Nexus++ with the PCIe bus on the FPGA board has been implemented, as well as a communication protocol to VSs; a light-weight runtime system that supports basic OmpSs tasking capabilities. The board can be inserted in the PCIe slot of any multicore machine and using a high level software API, OmpSs applications can be run using Nexus++.

In order to improve the performance of Nexus++, it has been thoroughly analyzed and improved in a take to parallize Nexus++ itself. Introducing a new distributed architecture of the hardware co-processor by having multiple task graphs working in parallel, the latest version of the co-processor, called Nexus#, further improves the scalability of OmpSs applications of fine-grain tasks or complex dependency patterns.

Besides implementing a low-latency task graph look-up mechanism using set associative cache-like structures, Nexus# uses a fast distribution algorithm that efficiently distributes the incoming memory addresses to the proper task graph. It also provides a configurable architecture in terms of the number of task graphs and the size of each task graph, which can be modified depending on the target system. Although targeting OmpSs applications, Nexus#'s low-latency retrieval task graphs can be used with other programming models.

Generating data and runtime traces for multiple benchmarks in the Starbench suite and embedding them in a Modelsim testbench, experimental results show that Nexus# achieves significant speedups for all the benchmarks on a 256-core pseudo-machine. Results also demonstrated that Nexus# outperforms Nanos, the official OmpSs runtime, as well as Nexus++ which has a central task graph manager, by orders of magnitude, for benchmarks that have very fine grain tasks and/or complex dependency patterns.

Finally, some future perspectives are presented, focusing on several modern SoC architectures, for which Nexus# can be suitable as a tightly coupled co-processor.

Bibliography

[1] IEEE Standard System C Language Reference Manual. *IEEE Std 1666-2005*, 2006. doi: 10.1109/IEEESTD.2006.99475.

[2] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Executing Task Graphs Using Work-Stealing. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010. doi: 10.1109/IPDPS.2010.5470403.

[3] Ghiath Al-Kadi and Andrei Sergeevich Terechko. A Hardware Task Scheduler for Embedded Video Processing. In *Proc. 4th Int. Conf. on High Performance Embedded Architectures and Compilers*, 2009. ISBN 978-3-540-92989-5. doi: 10.1007/978-3-540-92990-1_12.

[4] Altera. Stratix 10 SoC: Integrated High-Performance, Power-Efficient Processing. `https://www.altera.com/products/soc/portfolio/stratix-10-soc/overview.highResolutionDisplay.html`.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS'67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560.

[6] Michael Andersch, Chi Ching Chi, and Ben Juurlink. Programming Parallel Embedded and Consumer Applications in OpenMP Superscalar. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'12, pages 281–282, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145854.

[7] Michael Andersch, Chi Ching Chi, and Ben Juurlink. Using OpenMP Superscalar for Parallelization of Embedded and Consumer Applications. In *Proceed-*

ings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, 2012.

[8] ARM Ltd. The ARM926 Processor. `https://www.arm.com/products/processors/classic/arm9/arm926.php`.

[9] Oliver Arnold, Emil Matus, Benedikt Noethen, Markus Winter, Torsten Limberg, and Gerhard Fettweis. Tomahawk: Parallelism and Heterogeneity in Communications Signal Processing MPSoCs. *ACM Trans. Embed. Comput. Syst.*, 13 (3s):107:1–107:24, March 2014. ISSN 1539-9087. doi: 10.1145/2517087.

[10] Arvind, David August, Keshav Pingali, Derek Chiou, Resit Sendag, and Joshua J. Yi. Programming Multicores: Do Applications Programmers Need to Write Explicitly Parallel Programs? *IEEE Micro*, 30(3):19–33, May 2010. ISSN 0272-1732. doi: 10.1109/MM.2010.54.

[11] Supercomputing Technologies Group at MIT Laboratory for Computer Science. Cilk 5.4.6 Reference Manual. `http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf`, 1998.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, February 2011. ISSN 1532-0626. doi: 10.1002/cpe.1631.

[13] David I. August, Jialu Huang, Stephen R. Beard, Nick P. Johnson, and Thomas B. Jablin. Automatically Exploiting Cross-invocation Parallelism Using Runtime Information. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4673-5524-7. doi: 10.1109/CGO.2013.6495001.

[14] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The Design of OpenMP Tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, March 2009. ISSN 1045-9219. doi: 10.1109/TPDS.2008.105.

[15] Eduard Ayguadé, Rosa M. Badia, Daniel Jiménez, José R. Herrero, Jesús Labarta, Vladimir Subotic, and Gladys Utrera. Tareador: a tool to unveil parallelization strategies at undergraduate level. In *Proceedings of the Workshop on Computer Architecture Education*, WCAE '15, pages 1:1–1:8, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3717-5. doi: 10.1145/2795122.2795123.

[16] Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Javier Bueno, Alejandro Duran, Yoav Etsion, Montse Farreras, Roger Ferrer, Jesús Labarta, Vladimir Marjanovic, Lluis Martinell, Xavier Martorell, Josep M. Perez, Judit Planas, Alex Ramirez, Xavier Teruel, Ioanna Tsalouchidou, and Mateo Valero. Hybrid/Heterogeneous Programming with OmpSs and its Software/Hardware Implications. *Programming Multicore and Many-core Computing Systems*, 86:101, 2017.

[17] Evgenij Belikov, Pantazis Deligiannis, Prabhat Totoo, Malak Aljabri, and Hans-Wolfgang Loidl. A Survey of High-Level Parallel Programming Models. Technical Report HW-MACS-TR-0103, Department of Computer Science, Heriot-Watt University, December 2013.

[18] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesús Labarta. CellSs: A Programming Model for the Cell BE Architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188546.

[19] Pieter Bellens, Josep M. Perez, Felipe Cabarcas, Alex Ramirez, Rosa M. Badia, and Jesús Labarta. CellSs: Scheduling Techniques to Better Exploit Memory Hierarchy. *Sci. Program.*, 17(1-2):77–95, January 2009. ISSN 1058-9244.

[20] Micah J. Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova, and Andrew Brownsword. Synchronization via Scheduling: Techniques for Efficiently Managing Shared State. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 640–652, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993573.

[21] Eric Biscondi, Tom Flanagan, Frank Fruth, Zhihong Lin, and Filip Moerman. Maximizing Multicore Efficiency with Navigator Runtime. *White Paper, feb*, 2012.

[22] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM. ISBN 0-89791-700-6. doi: 10.1145/209936.209958.

[23] OpenMP Architecture Review Board. OpenMP Application Program Interface v.4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, 2013.

[24] Steffen Brinkmann, José Gracia, Christoph Niethammer, and Rainer Keller. TEMANEJO – a debugger for task based parallel programming models. In *Applications, Tools and Techniques on the Road to Exascale Computing, Proceedings of the conference ParCo 2011, 31 August - 3 September 2011, Ghent, Belgium*, pages 639–645, 2011. doi: 10.3233/978-1-61499-041-3-639.

[25] Steffen Brinkmann, José Gracia, and Christoph Niethammer. *Task Debugging with TEMANEJO*, pages 13–21. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-37349-7. doi: 10.1007/978-3-642-37349-7_2.

[26] Barcelona Supercomputing Center BSC. Cell Superscalar. `http://www.bsc.es/plantillaG.php?cat_id=179`, 2009.

[27] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Productive Cluster Programming with OmpSs. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, pages 555–566, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23399-9.

[28] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, RosaM. Badia, Eduard Ayguadé, and Jesús Labarta. Productive cluster programming with ompss. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6852 of *Lecture Notes in Computer Science*, pages 555–566. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-23399-9. doi: 10.1007/978-3-642-23400-2_52.

[29] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel*

Comput., 35(1):38–53, January 2009. ISSN 0167-8191. doi: 10.1016/j.parco.2008.10.002.

[30] Simone Campanoni, Timothy Jones, Glenn Holloway, Gu-Yeon Wei, and David Brooks. The HELIX Project: Overview and Directions. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 277–282, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1199-1. doi: 10.1145/2228360.2228412.

[31] Marc Casas, Miquel Moreto, Lluc Alvarez, Emilio Castillo, Dimitrios Chasapis, Timothy Hayes, Luc Jaulmes, Oscar Palomar, Osman Unsal, Adrian Cristal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, chapter Runtime-Aware Architectures, pages 16–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-48096-0. doi: 10.1007/978-3-662-48096-0_2.

[32] Jerónimo Castrillón. *Programming heterogeneous MPSoCs : Tool Flows to Close the Software Productivity Gap*. PhD thesis, Aachen, 2013. URL `http://publications.rwth-aachen.de/record/211242`. Aachen, Techn. Hochsch., Diss., 2013.

[33] Jerónimo Castrillón, Diandian Zhang, Torsten Kempf, Bart Vanthournout, Rainer Leupers, and Gerd Ascheid. Task Management in MPSoCs: An ASIP Approach. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 587–594, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-800-1. doi: 10.1145/1687399.1687508.

[34] Jerónimo Castrillón, Rainer Leupers, and Gerd Ascheid. MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs. *IEEE Transactions on Industrial Informatics*, 9(1):527–545, Feb 2013. ISSN 1551-3203. doi: 10.1109/TII.2011.2173941.

[35] Jianjiang Ceng, Jerónimo Castrillón, Weihua Sheng, H Scharwachter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tsuyoshi Isshiki, and Hiroaki Kunieda. MAPS: An Integrated Framework for MPSoC Application Parallelization. In *2008 45th ACM/IEEE Design Automation Conference*, pages 754–759, June 2008. doi: 10.1145/1391469.1391663.

[36] Barcelona Supercomputing Center. Extrae Project Website. `https://tools.bsc.es/extrae`, .

[37] Barcelona Supercomputing Center. Nanos++ Project Website. `https://pm.bsc.es/nanox`, .

[38] Barcelona Supercomputing Center. Paraver: a flexible performance analysis tool. `https://tools.bsc.es/paraver`, .

[39] BSC-CNS | Barcelona Supercomputing Center. OmpSs Examples and Exercises. `https://pm.bsc.es/ompss-docs/examples/index.html`, 2016.

[40] Ernie Chan, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. Supermatrix Out-of-order Scheduling of Matrix Operations for SMP and Multi-core Architectures. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 116–125, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-667-7. doi: 10.1145/1248377.1248397.

[41] L. Chang, D. J. Frank, R. K. Montoye, S. J. Koester, B. L. Ji, P. W. Coteus, R. H. Dennard, and W. Haensch. Practical strategies for power-efficient computing technologies. *Proceedings of the IEEE*, 98(2):215–236, Feb 2010. ISSN 0018-9219. doi: 10.1109/JPROC.2009.2035451.

[42] Chi Ching Chi, Ben Juurlink, and Cor Meenderinck. Evaluation of Parallel H.264 Decoding Strategies for the Cell Broadband Engine. In *Proc. 24th ACM Int. Conf. on Supercomputing*, 2010. ISBN 978-1-4503-0018-6. doi: 10.1145/1810085.1810102.

[43] Intel Corporation. Intel® Cilk™ Plus Project Website. `https://www.cilkplus.org`.

[44] Ron Cytron. Doacross: Beyond Vectorization for Multiprocessors. In *International Conference on Parallel Processing, ICPP'86*, pages 836–844, 1986.

[45] Leonardo Dagum and Ramesh Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *IEEE Computational Sci. Eng.*, 1998. ISSN 1070-9924. doi: 10.1109/99.660313.

[46] Tamer Dallou, Divino Cesar Soares Lucas, Guido Araujo, Lucas Morais, Eduardo Ferreira Barbosa, Michael Frank, Richard Bagley, and Raj Sayana. Task Parallel Programming Model + Hardware Acceleration = Performance Advantage. `https://www.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.24-Posters-Epub/HC28.24.100-71_lge_HotChipsPoster-Final.pdf`, 2016.

[47] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: Recording Microprocessor History. *ACM Queue*, 10(4): 10:10–10:27, April 2012. ISSN 1542-7730. doi: 10.1145/2181796.2181798.

[48] Technische Universität Darmstadt. DiscoPoP (Discovery of Potential Parallelism) Project Website. `https://www.parallel.informatik.tu-darmstadt.de/multicore-group/discopop/`.

[49] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th Symp. on Operating Systems Design & Implementation*, 2004.

[50] Robert H. Dennard, Fritz H. Gaensslen, Hwa nien Yu, V. Leo Rideout, Ernest Bassous, Andre, and R. Leblanc. Design of ion-implanted mosfets with very small physical dimensions. *IEEE J. Solid-State Circuits*, page 256, 1974.

[51] Andi Drebes. Aftermath Project Website. `https://www.aftermath-tracing.com/`.

[52] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach-Temam. Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. In *7th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG, associated with HiPEAC), Vienna, Austria*, 2014.

[53] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 125–137, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4121-9. doi: 10.1145/2967938.2967946.

[54] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011. doi: 10.1142/S0129626411000151.

[55] Nina Engelhardt, Tamer Dallou, Ahmed Elhossini, and Ben Juurlink. An Integrated Hardware-Software Approach to Task Graph Management. In *16th IEEE International Conference on High Performance and Communications HPCC-2014*, pages 392–399, 2014. ISBN 978-1-4799-6123-8. doi: 10.1109/HPCC.2014.66.

[56] Yoav Etsion, Alex Ramirez, Rosa M. Badia, and Jesús labarta. Cores as Functional Units: A Task-Based, Out-of-Order, Dataflow Pipeline. In *Proc. Int. Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*, 2009.

[57] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Task Superscalar: An Out-of-Order Task Pipeline. *IEEE/ACM Int. Symposium on Microarchitecture*, 2010. ISSN 1072-4451. doi: 10.1109/MICRO.2010.13.

[58] Yoav Etsion, Alex Ramirez, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Task Superscalar: Using Processors as Functional Units. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 16–16, Berkeley, CA, USA, 2010. USENIX Association.

[59] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071.

[60] Alcides Fonseca, Bruno Cabral, João Rafael, and Ivo Correia. Automatic Parallelization: Executing Sequential Programs on a Task-Based Parallel Runtime. *International Journal of Parallel Programming*, 44(6):1337–1358, December 2016. ISSN 0885-7458. doi: 10.1007/s10766-016-0426-5.

[61] Foundation for Research, Technology Hellas (FORTH), Institute of Computer Science (ICS), and Computer Architecture & VLSI Systems Labora-

tory (CARV). The Myrmics Runtime System Project Website. `http://www.myrmics.com/`.

[62] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. *SIGPLAN Not.*, 33(5):212–223, May 1998. ISSN 0362-1340. doi: 10.1145/277652.277725.

[63] Thierry Gautier, Fabien Lementec, Vincent Faucher, and Bruno Raffin. X-kaapi: A Multi Paradigm Runtime for Multicore Architectures. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 728–735, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5117-3. doi: 10.1109/ICPP.2013.86.

[64] Thierry Gautier, Joao V. F. Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 1299–1308, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4971-2. doi: 10.1109/IPDPS.2013.66.

[65] Roberto Giorgi, Rosa M. Badia, François Bodin, Albert Cohen, Paraskevas Evripidou, Paolo Faraboschi, Bernhard Fechner, Guang R. Gao, Arne Garbade, Rahul Gayatri, Sylvain Girbal, Daniel Goodman, Behran Khan, Souad Koliaï, Joshua Landwehr, Nhat Minh Lê, Feng Li, Mikel Luján, Avi Mendelson, Laurent Morin, Nacho Navarro, Tomasz Patejko, Antoniu Pop, Pedro Trancoso, Theo Ungerer, Ian Watson, Sebastian Weis, Stéphane Zuckerman, and Mateo Valero. Teraflux: Harnessing dataflow in next generation teradevices. *Microprocess. Microsyst.*, 38(8):976–990, November 2014. ISSN 0141-9331. doi: 10.1016/j.micpro.2014.04.001.

[66] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchial CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 107–121, New York, NY, USA, 1996. ACM. ISBN 1-880446-82-0. doi: 10.1145/238721.238766.

[67] Ananth Grama. *Introduction to Parallel Computing*. Pearson Education. Addison-Wesley, 2003. ISBN 9780201648652.

[68] Kate Gregory and Ade Miller. *C++ AMP: Accelerated Massive Parallelism with Microsoft® Visual C++®*. Developer Reference. Microsoft Press, 2012. ISBN 9780735668195.

[69] J. P. Grossman, Jeffrey S. Kuskin, Joseph A. Bank, Michael Theobald, Ron O. Dror, Douglas J. Ierardi, Richard H. Larson, U. Ben Schafer, Brian Towles, Cliff Young, and David E. Shaw. Hardware Support for Fine-grained Event-driven Computation in Anton 2. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 549–560, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451175.

[70] Gagan Gupta and Gurindar S. Sohi. Dataflow Execution of Sequential Imperative Programs on Multicore Architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 59–70, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1053-6. doi: 10.1145/2155620.2155628.

[71] Haopeng Han. Evaluating Nexus++ as a Tightly Coupled Coprocessor for ARM Multi-Core Processors. Master's thesis, Technische Universität Berlin, 2015.

[72] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003. ISBN 1558607242.

[73] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. ISBN 0-8186-3810-9. doi: 10.1145/165123.165164.

[74] The High Performance Computing Center Stuttgart (HLRS). TEMANEJO Project Website. `https://www.hlrs.de/de/solutions-services/service-portfolio/programming/hpc-development-tools/temanejo/`.

[75] Jan Hoogerbrugge and Andrei Terechko. A Multithreaded Multicore System for Embedded Media Processing. *Trans. on High-Performance Embedded Architectures and Compilers*, 3(2), 2008.

[76] Ali R. Hurson, Joford T. Lim, Krishna M. Kavi, and Ben Lee. Parallelization of DOALL and DOACROSS loops - A survey. *Advances in Computers*, 45:53–103, 1997. doi: 10.1016/S0065-2458(08)60706-8.

[77] An Huynh, Douglas Thain, Miquel Pericàs, and Kenjiro Taura. DAGViz: A DAG Visualization Tool for Analyzing Task-parallel Program Traces. In *Proceedings of the 2Nd Workshop on Visual Performance Analysis*, VPA '15, pages 3:1–3:8, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4013-7. doi: 10.1145/2835238.2835241.

[78] IBM. IBM Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems. `http://www-304.ibm.com/webapp/set2/sas/f/capi/CAPI_POWER8.pdf`, 2014.

[79] Paolo Ienne and Rainer Leupers. *Customizable Embedded Processors: Design Technologies and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 0123695260, 9780080490984.

[80] Advanced Micro Devices Inc. AMD64 Architecture Programmer's Manual Volume 2: System Programming. http://support.amd.com/TechDocs/24593.pdf, March 2017.

[81] Texas Instruments. Multicore DSP+ ARM KeyStone II System-on-Chip (SoC). *Retrieved September*, 9:2016, 2013.

[82] Kazuhisa Ishizaka, Motoki Obata, and Hironori Kasahara. *Coarse-grain Task Parallel Processing Using the OpenMP Backend of the OSCAR Multigrain Parallelizing Compiler*, pages 457–470. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-39999-5. doi: 10.1007/3-540-39999-2_43.

[83] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A Scalable Architecture for Ordered Parallelism. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 228–241, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4034-2. doi: 10.1145/2830772.2830777.

[84] Hironori Kasahara, Motoki Obata, and Kazuhisa Ishizaka. Automatic Coarse Grain Task Parallel Processing on SMP using OpenMP. In *International*

Workshop on Languages and Compilers for Parallel Computing, pages 189–207. Springer Berlin Heidelberg, 2000.

[85] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on Parallel Programming Model. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, NPC'08, pages 266–275, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88139-1. doi: 10.1007/978-3-540-88140-7_24.

[86] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1-55860-286-0.

[87] Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl/`.

[88] Keiji Kimura, Yasutaka Wada, Hirufumi Nakano, Takeshi Kodaka, Jun Shirako, Kazuhisa Ishizaka, and Hironori Kasahara. Multigrain Parallel Processing on Compiler Cooperative Chip Multiprocessor. In *Proceedings of the 9th Annual Workshop on Interaction Between Compilers and Computer Architectures*, INTERACT '05, pages 11–20, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2321-8. doi: 10.1109/INTERACT.2005.9.

[89] A.D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011. ISBN 9781139470315.

[90] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How Much Parallelism is There in Irregular Applications? In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 3–14, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6. doi: 10.1145/1504176.1504181.

[91] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *Proc. 34th Annual Int. Symp. on Computer Architecture*, 2007. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250683.

[92] Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. Data-Driven Multithreading Using Conventional Microprocessors. *IEEE Trans. Parallel Distrib.*

Syst., 17:1176–1188, October 2006. ISSN 1045-9219. doi: 10.1109/TPDS.2006.136.

[93] HP Laboratories. CACTI 5.3. `http://www.hpl.hp.com/research/cacti/`.

[94] Leslie Lamport. The Parallel Execution of DO Loops. *Commun. ACM*, 17(2): 83–93, February 1974. ISSN 0001-0782.

[95] Simon H. Lavington. *A History of Manchester Computers*. British Computer Society, 1998. ISBN 9781902505015.

[96] Charles E. Leiserson. The Cilk++ Concurrency Platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-497-3. doi: 10.1145/1629911.1630048.

[97] Rainer Leupers and Jerónimo Castrillón. MPSoC Programming Using the MAPS Compiler. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, pages 897–902, Piscataway, NJ, USA, 2010. IEEE Press. ISBN 978-1-60558-837-7.

[98] Zhen Li, Rohit Atre, Zia Ul Huda, Ali Jannesari, and Felix Wolf. Unveiling parallelization opportunities in sequential programs. *Journal of Systems and Software*, 117:282–295, July 2016. doi: 10.1016/j.jss.2016.03.045.

[99] Torsten Limberg, M Winter, M Bimberg, R Klemm, M Tavares, H Ahlendorf, E Matúš, G Fettweis, H Eisenreich, G Ellguth, et al. A Heterogeneous MPSoC with Hardware supported Dynamic Task Scheduling for Software Defined Radio. In *Design Automation Conference*, volume 2009, pages 97–98, 2009.

[100] M Lippett. An IP Core Based Approach to the On-Chip Management of Heterogeneous SoCs. *IP Based SoC Design Forum and Exhibition*, 2004, 2004.

[101] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043587.

[102] Dumitrel Loghin, Bogdan Marius Tudor, and Yong Meng Teo. An Approach for Direct Dataflow Execution on Contemporary Multicore Systems. In *Data-Flow Execution Models for Extreme Scale Computing*, pages 1–8, Sept 2013. doi: 10.1109/DFM.2013.9.

[103] Divino Cesar S. Lucas and Guido Araujo. The Batched DOACROSS Loop Parallelization Algorithm. In *2015 International Conference on High Performance Computing Simulation (HPCS)*, pages 476–483, July 2015. doi: 10.1109/HPCSim.2015.7237079.

[104] Spyros Lyberis. *Myrmics: A Scalable Runtime System for Global Address Spaces.* PhD thesis, PhD thesis, University of Crete, 2013.

[105] Spyros Lyberis, Polyvios Pratikakis, Iakovos Mavroidis, and Dimitrios S Nikolopoulos. Myrmics: Scalable, Dependency-aware Task Scheduling on Heterogeneous Manycores. *arXiv preprint arXiv:1606.04282*, 2016.

[106] Chris A. Mack. Fifty Years of Moore's Law. *IEEE Transactions on Semiconductor Manufacturing*, 24(2):202–207, May 2011. ISSN 0894-6507. doi: 10.1109/TSM.2010.2096437.

[107] George Matheou and Paraskevas Evripidou. Architectural Support for Data-Driven Execution. *ACM Trans. Archit. Code Optim.*, 11(4):52:1–52:25, January 2015. ISSN 1544-3566. doi: 10.1145/2686874.

[108] C.H. Meenderinck. *Improving the Scalability of Multicore Systems, with a Focus on H.264 Video Decoding.* PhD thesis, Delft University of Technology, 2010.

[109] Cor Meenderinck and Ben Juurlink. A Case for Hardware Task Management Support for the StarSs Programming Model. In *Proc. 13th Euromicro Conf. on Digital System Design: Architectures, Methods and Tools*, 2010. doi: 10.1109/DSD.2010.63. Sp. Session on Multicore Systems: Des. and Apps.

[110] Cor Meenderinck, Arnaldo Azevedo, Mauricio Alvarez, Ben Juurlink, and Alex Ramirez. Parallel scalability of H.264. In *Proceedings of the first Workshop on Programmability Issues for Multi-Core Computers*, 2008.

[111] Cor Meenderinck, Arnaldo Azevedo, Ben Juurlink, Mauricio Alvarez Mesa, and Alex Ramirez. Parallel Scalability of Video Decoders. *J. Signal Process. Syst.*, 57:173–194, November 2009. ISSN 1939-8018. doi: 10.1007/s11265-008-0256-9.

[112] Hiroki Mikami, Shumpei Kitaki, Masayoshi Mase, Akihiro Hayashi, Mamoru Shimaoka, Keiji Kimura, Masato Edahiro, and Hironori Kasahara. *Evaluation of Power Consumption at Execution of Multiple Automatically Parallelized and Power Controlled Media Applications on the RP2 Low-Power Multicore*, pages 31–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-36036-7. doi: 10.1007/978-3-642-36036-7_3.

[113] Ingo Molnar. The Native POSIX Thread Library for Linux. Technical report, RedHat, Inc, 2003.

[114] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[115] NVIDIA. CUDA - Compute Unified Device Architecture. `https://developer.nvidia.com/cuda-zone`.

[116] NVIDIA. NVIDIA Launches Tegra X1 Mobile Super Chip. `http://nvidianews.nvidia.com/news/nvidia-launches-tegra-x1-mobile-super-chip`, 2015.

[117] Motoki Obata, Kazuhisa Ishizaka, and Hironori Kasahara. Automatic Coarse Grain Task Parallel Processing Using OSCAR Multigrain Parallelizing Compiler. In *Ninth International Workshop on Compilers for Parallel Computers (CPC 2001)*, 2001.

[118] University of Campinas and LG Electronics. The Multicore Task Scheduling Platform Project Website. `http://www.mtsplib.org/`.

[119] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2440-0. doi: 10.1109/MICRO.2005.13.

[120] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011. ISBN 9780123742605.

[121] Sanggyu Park, Do sun Hong, and Soo-Ik Chae. A hardware operating system kernel for multi-processor systems. *IEICE Electronics Express*, 5(9):296–302, 2008. doi: 10.1587/elex.5.296.

[122] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013. ISBN 0124077269, 9780124077263.

[123] Josep M. Perez, Rosa M. Badia, and Jesús Labarta. A Dependency-Aware Task-Based Programming Environment for Multi-core Architectures. In *2008 IEEE International Conference on Cluster Computing*, pages 142–151, Sept 2008. doi: 10.1109/CLUSTR.2008.4663765.

[124] Josep M. Perez, Rosa M. Badia, and Jesús Labarta. Handling Task Dependencies Under Strided and Aliased References. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 263–274, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0018-6. doi: 10.1145/1810085.1810122.

[125] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, 2005. doi: 10.1109/ISSCC.2005.1493930.

[126] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The Tao of Parallelism in Algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993501.

[127] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical Task-Based Programming With StarSs. *International Journal of High Performance Computing Applications*, 2009. ISSN 1094-3420. doi: 10.1177/1094342009106195.

[128] Artur Podobas, Mats Brorsson, and Vladimir Vlassov. TurboBŁYSK: Scheduling for Improved Data-Driven Task Performance with Fast Dependency Resolution. In *Using and Improving OpenMP for Devices, Tasks, and More*, volume 8766 of *Lecture Notes in Computer Science*, pages 45–57. Springer International Publishing, 2014. ISBN 978-3-319-11453-8. doi: 10.1007/978-3-319-11454-5_4.

[129] Antoniu Pop and Albert Cohen. OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs. *ACM Trans. Archit. Code Optim.*, 9(4):53:1–53:25, January 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400712.

[130] Power.org. Power.org Embedded Bus Architecture Report. `www.power.org/resources/downloads/Embedded_Bus_Arch_Report_1.0.pdf`, 2008.

[131] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 1–11, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993500.

[132] SARC European Project. Parallel Programming Models for Heterogeneous Multicore Architectures. *Micro, IEEE*, 30(5):42 –53, sept.-oct. 2010. ISSN 0272-1732. doi: 10.1109/MM.2010.94.

[133] Nikola Rajovic, Paul M. Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 40:1–40:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503281.

[134] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. Parcae: A System for Flexible Parallel Execution. In *Proceedings of the 33rd ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '12, pages 133–144, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254082.

[135] A. Ramirez, F. Cabarcas, B. Juurlink, M. Alvarez Mesa, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, and G. Gaydadjiev. The SARC Architecture. *Micro, IEEE*, 30(5):16 –29, sept.-oct. 2010. ISSN 0272-1732. doi: 10.1109/MM.2010.79.

[136] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., 1st edition, 2007. ISBN 9780596514808.

[137] Karl Rupp. 40 Years of Microprocessor Trend Data. `https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/`, 2015.

[138] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS'10, pages 311–322, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-839-1. doi: 10.1145/1736020.1736055.

[139] Hendrik Seidel. *A Task Level Programmable Processor*. WiKu-Verlag, 2006. ISBN 9783865531704.

[140] Semiconductor Industry Associations of the United States, Europe, Japan, South Korea and Taiwan. The International Technology Roadmap for Semiconductors 2013 Edition. `http://www.itrs2.net/2013-itrs.html`, 2013.

[141] Semiconductor Industry Associations of the United States, Europe, Japan, South Korea and Taiwan. The International Technology Roadmap for Semiconductors 2015 Edition. `http://www.itrs2.net/itrs-reports.html`, 2015.

[142] David E. Shaw, J. P. Grossman, Joseph A. Bank, Brannon Batson, J. Adam Butts, Jack C. Chao, Martin M. Deneroff, Ron O. Dror, Amos Even, Christopher H. Fenton, Anthony Forte, Joseph Gagliardo, Gennette Gill, Brian Greskamp, C. Richard Ho, Douglas J. Ierardi, Lev Iserovich, Jeffrey S. Kuskin, Richard H. Larson, Timothy Layman, Li-Siang Lee, Adam K. Lerer, Chester Li, Daniel Killebrew, Kenneth M. Mackenzie, Shark Yeuk-Hai Mok, Mark A.

Moraes, Rolf Mueller, Lawrence J. Nociolo, Jon L. Peticolas, Terry Quan, Daniel Ramot, John K. Salmon, Daniele P. Scarpazza, U. Ben Schafer, Naseer Siddique, Christopher W. Snyder, Jochen Spengler, Ping Tak Peter Tang, Michael Theobald, Horia Toma, Brian Towles, Benjamin Vitale, Stanley C. Wang, and Cliff Young. Anton 2: Raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 41–53, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.9.

[143] Magnus Själander, Andrei Terechko, and Marc Duranton. A Look-Ahead Task Management Unit for Embedded Multi-Core Architectures. In *Proc. 11th EUROMICRO Conf. on Digital System Design: Architectures, Methods and Tools*, 2008. doi: 10.1109/DSD.2008.45.

[144] Yan Solihin. *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall/CRC, 1st edition, 2016. ISBN 1482211181, 9781482211184.

[145] Statista.com. Quarterly personal computer (PC) vendor shipments worldwide, from 2009 to 2016, by vendor. `https://www.statista.com/statistics/263393`, 2016.

[146] Statista.com. Global smartphone shipments from 2009 to 2016. `https://www.statista.com/statistics/271491`, 2016.

[147] Statista.com. Global tablet unit shipments by vendor from 3rd quarter 2011 to 4th quarter 2016. `https://www.statista.com/statistics/276651`, 2016.

[148] Xubin Tan, Jaume Bosch, Daniel Jiménez-González, Carlos Álvarez Martínez, Eduard Ayguadé, and Mateo Valero. Performance analysis of a hardware accelerator of dependence management for task-based dataflow programming models. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 225–234, April 2016. doi: 10.1109/ISPASS.2016.7482097.

[149] Sagnak Tasirlar and Vivek Sarkar. Data-Driven Tasks and Their Implementation. In *Proceedings of the 2011 International Conference on Parallel Processing*,

ICPP'11, pages 652–661, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4510-3. doi: 10.1109/ICPP.2011.87.

[150] Enric Tejedor, Montse Farreras, David Grove, Rosa M. Badia, Gheorghe Almasi, and Jesús Labarta. ClusterSs: A Task-based Programming Model for Clusters. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 267–268, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0552-5. doi: 10.1145/1996130.1996168.

[151] The Economist. The Economist: The End of Moore's Law. `http://www.economist.com/blogs/economist-explains/2015/04/economist-explains-17`, 2015.

[152] D. Theodoropoulos, D. Pnevmatikatos, C. Alvarez, E. Ayguade, J. Bueno, A. Filgueras, D. Jimenez-Gonzalez, X. Martorell, N. Navarro, C. Segura, C. Fernandez, D. Oro, J. R. Saeta, P. Gai, A. Rizzo, and R. Giorgi. The AXIOM project (Agile, eXtensible, fast I/O Module). In *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 262–269, July 2015. doi: 10.1109/SAMOS.2015.7363684.

[153] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P Jouppi. Cacti 5.1. Technical report, Technical Report HPL-2008-20, HP Labs, 2008. URL `http://www.hpl.hp.com/techreports/2008/HPL-2008-20.pdf`.

[154] Top 500. The 500 Most Powerful Commercially Available Computer Systems. `http://www.top500.org/`.

[155] Top 500. Sunway TaihuLight: National Supercomputing Center in Wuxi. `https://www.top500.org/resources/top-systems/sunway-taihulight-national-supercomputing-center-i/`, June 2016.

[156] Erik B. van der Tol, Egbert G. Jaspers, and Rob H. Gelderblom. Mapping of H.264 Decoding on a Multiprocessor Architecture. In *Proc. SPIE Conf. on Image and Video Communications and Processing*, 2003. doi: 10.1117/12.476234.

[157] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. In *Proceedings of*

the 19th international conference on Parallel architectures and compilation techniques, PACT '10, pages 389–400, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi: 10.1145/1854273.1854322.

[158] Hans Vandierendonck, Polyvios Pratikakis, and Dimitrios S Nikolopoulos. Parallel Programming of General-Purpose Programs Using Task-Based Programming Models. In *3rd USENIX workshop on Hot Topics in Parallelism (HotPar 2011)*. USENIX Association, 2011.

[159] Hans Vandierendonck, George Tzenakis, and Dimitrios S. Nikolopoulos. A Unified Scheduler for Recursive and Task Dataflow Parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 1–11, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4566-0. doi: 10.1109/PACT.2011.7.

[160] Hans Vandierendonck, George Tzenakis, and Dimitrios S. Nikolopoulos. Analysis of Dependence Tracking Algorithms for Task Dataflow Execution. *ACM Transactions on Architecture and Code Optimization*, 10(4):61:1–61:24, December 2013. ISSN 1544-3566. doi: 10.1145/2555289.2555316.

[161] Marinus Veldhorst. Gaussian Elimination with Partial Pivoting on an MIMD Computer. *Journal of Parallel and Distributed Computing*, 1989. ISSN 0743-7315. doi: 10.1016/0743-7315(89)90042-7.

[162] Chao Wang, Xi Li, Junneng Zhang, Peng Chen, Xiaojing Feng, and Xuehai Zhou. FPM: A flexible programming model for MPSoC on FPGA. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 477–484. IEEE, 2012. doi: 10.1109/IPDPSW.2012.62.

[163] Chao Wang, Xi Li, Junneng Zhang, Xuehai Zhou, and Xiaoning Nie. MP-Tomasulo: A Dependency-Aware Automatic Parallel Execution Engine for Sequential Programs. *ACM Trans. Archit. Code Optim.*, 10(2):9:1–9:26, May 2013. ISSN 1544-3566. doi: 10.1145/2459316.2459320.

[164] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985. ISBN 0-201-08222-5.

[165] Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the H.264 / AVC Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003. ISSN 1051-8215. doi: 10.1109/TCSVT.2003.815165.

[166] Wikipedia. Intel 80386. `http://en.wikipedia.org/w/index.php?title=Intel%2080386&oldid=746554192`, 2016.

[167] Wikipedia. List of CPU power dissipation figures. `http://en.wikipedia.org/w/index.php?title=List%20of%20CPU%20power%20dissipation%20figures&oldid=747798545`, 2016.

[168] Wikipedia. Transistor count. `http://en.wikipedia.org/w/index.php?title=Transistor%20count&oldid=746450017`, 2016.

[169] Xilinx. Endpoint Block Plus v1.15 for PCI Express User Guide. `http://www.xilinx.com/support/documentation/ip_documentation/pcie_blk_plus/v1_15/pcie_blk_plus_ug341.pdf`, 2011.

[170] Xilinx. ML505/ML506/ML507 Evaluation Platform User Guide. `www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf`, 2011.

[171] Xilinx. LogiCORE IP FIFO Generator v9.3. `http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v9_3/pg057-fifo-generator.pdf`, 2012.

[172] Xilinx. UltraScale Architecture and Product Overview. `http://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf`, 2015.

[173] Fahimeh Yazdanpanah, Daniel Jimenez-Gonzalez, Carlos Alvarez-Martinez, Yoav Etsion, and Rosa M. Badia. Analysis of the Task Superscalar Architecture Hardware Design. *Procedia Computer Science*, 18:339 – 348, 2013. ISSN 1877-0509. doi: 10.1016/j.procs.2013.05.197.

[174] Fahimeh Yazdanpanah, Daniel Jimenez-Gonzalez, Carlos Alvarez-Martinez, Yoav Etsion, and Rosa M. Badia. FPGA-Based Prototype of the Task Superscalar Architecture. In *7th HiPEAC Workshop on Reconfigurable Computing (WRC 2013)*, 2013. doi: 10.1109/ISPASS.2011.5762718.

[175] Fahimeh Yazdanpanah, Carlos Álvarez, Daniel Jiménez-González, Rosa M. Badia, and Mateo Valero. Picos: A hardware runtime architecture support for OmpSs. *Future Generation Computer Systems*, 2015. ISSN 0167-739X. doi: 10.1016/j.future.2014.12.010.

[176] Diandian Zhang, Jeronimo Castrillon, Stefan Schürmans, Gerd Ascheid, Rainer Leupers, and Bart Vanthournout. System-Level Analysis of MPSoCs with a Hardware Scheduler. In *Advancing Embedded Systems and Real-Time Communications with Emerging Technologies*, pages 335–367. IGI Global, 2014. doi: 10.4018/978-1-4666-6034-2.ch014.