HARDWARE RUNTIME MANAGEMENT FOR TASK-BASED PROGRAMMING MODELS

Xubin Tan

Barcelona, 2018

ADVISORS:

Carlos Álvarez Martínez
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center

Daniel Jiménez-González
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center

A thesis submitted in fulfillment of
the requirements for the degree of
DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Acknowledgments

This thesis has been a long, difficult and enriching journey which has been possible, thanks to the guidance and support of many people to whom I am very grateful.

First and foremost I would like to thank my advisors Carlos Alvarez, Daniel Jimenez. Carlos and Daniel have guided me through the inspiring and also difficult years of this journey and have taught me many lessons that now define me as a researcher. Among many things, from them I have learned to believe in my ideas, to stay focused, to push hard for my goals, to accept criticism, to not get distressed when progress is stalled by obstacles, and to continue when the problems are solved. I want to sincerely thank Carlos and Dani and give them credit for all the effort and time and encouragement they have contributed to my studies, for the confidence they have placed in me and my work and, more importantly, for embracing me as their student.

Next, I would like to thank many people from DAC and BSC, that have helped me during my PhD. Mateo Valero, who offered me a chance in this amazing research center, which opened the door for me to the research world. Eduard Ayguade for being an inspiring scholar, for always having a sense of humor and a positive attitude, which cheers up the people who work with him. Jesus Labarta, for being an excellent and patient teacher in the class for performance analysis using Paraver and Dimemas. Jaume Bosch, who has been cooperating with my project on the last three years, without whom I could not have made it this far. Antonio Filgueras and Miquel Vidal, who are always very patient with all my naive questions. All the members in the RoMoL team, Miquel Moretó, Marc Casas, Cesar Állande, Vladimir Dimić, Luc Jaulmes, Dimitris Chasapis, Lluc Alvarez, Calvin Bulla, Constantino Gomez, Francesc Martinez, Isaac Sachez, Adrian Barredo, Helena Caminal, and all the others. Each of them is unique, knowledgeable and fun in many ways, and many of them have been very encouraging and helpful. They have created a wonderful environment, and I could not have asked for more to work and study.

I would also like to thank the professors in my pre-defense committee, Jesus Labarta, Rosa Badia and Josep Llosa, who have been very helpful and given me many valuable

suggestions.

Finally I would like to thank my family, my father Tan Siwei, my mother Chen Nian'er, my brother Tan Xuming and my niece Tan Muyu. They are the ones that give me life, show me love and are forever there for me. They are the ones that teach me the difficulties but also the beauty of life. I am a very lucky person to have them.

Abstract

Task-based programming models allow programmers to express applications as a collection of tasks with dependences. They are simple to use and greatly improve programmability by using software runtimes to exploit task parallelism and heterogeneity over multicore, many-core and heterogeneous platforms. In these programming models, the runtimes guarantee correct execution order by managing tasks using task-dependency graphs (TDGs). These runtimes are powerful enough to provide high performance with coarse-grained tasks although they impose oveheads on the application execution to maintain all the information they need to do their work. However, as the current trend in processor architectures keeps including more cores and heterogeneity (in fact complexity) in the system, coarse-grained parallelism is not enough to feed all the underlying resources. Instead, fine-grained tasks are preferable as they are able to expose higher parallelism in applications but the overhead introduced by the software runtimes under these conditions prevent an efficient exploitation of fine-grained parallelism. The two most critical runtime overheads are task dependency graph management and task scheduling to heterogeneous systems.

There is no doubt that software is of great importance, as it allows for great expressivity and flexibility. However, it is also unquestionable that hardware is known for higher speed and energy-efficient designs. Therefore, we propose a hardware architecture, Picos, consisting of a hardware task dependence manager including nested task support, and a heterogeneous task scheduler, to accelerate the critical runtime functions for task-based programming models. With Picos, we aim at extending the benefit of these programming models into exploiting fine-grained task parallelism and heterogeneity.

As a proof-of-concept, in this work Picos has been designed in VHDL and implemented in a System-on-chip platform consisting of regular ARM SMP cores and an integrated FPGA. Three prototypes of Picos have been designed, developed and analyzed with real benchmarks. Picos designs have been connected to the SMP processors and numerous HW functional accelerators (Hardware specialized task execution units). In addition,

the designs have been integrated with a state-of-the-art task-based parallel programming model, running in a Linux system.

Performance and energy consumption results have been obtained with real executions in real hardware platforms and compared to the results using a task-based parallel programming with a software-only runtime. These results show that our proposed runtime system obtains better performance with a lower energy consumption than the software-only alternative. With 4 threads and up to 4 HW functional accelerators, it achieved up to 7.6x speedup and up to 90% of energy savings with real benchmarks comparing against the software-only runtime. In addition, with 4 threads and 12 accelerators, it gains up to 16.2x speedup with real applications when compared with the software-only runtime. The trend of obtaining more benefits when there are more system resources (in terms of both their core count and heterogeneity) leads us to believe that the impact of using a hardware accelerated runtime will be even more significant than a software-only alternative in bigger systems than the ones evaluated.

This research work has also opened several future directions that are worth exploring. Although currently only SMP and FPGA have been explored due to the limited hardware available, Picos is not restricted to these platforms, as it is designed to be adaptable to manage a great variety of hardware such as ASICs, GPUs, Big-little type architectures, etc. In addition, Picos runtime can be easily adapted to support other task-based programming models; as the main three functions - task dependency graph management, nested task support and heterogeneous task scheduling - are common characteristics of nearly all of them. Finally another interesting proposed path is the use of Picos to manage the sleep/wake-up of cores and other hardware execution units. During this work, we discovered that for many applications, the hardware resources are kept on working normally when there is not enough parallelism available. A hardware manager like the one proposed can be used for a quick hardware sleep/wake-up mechanism leading to a large amount of time savings without reducing the performance.

Contents

Chapter 1

Introduction

For many years, the prediction of Moore's law that the number of transistors in a dense integrated circuit doubles about every two years has been accurate, and the semiconductor tecnology has droven the progress of every new generation of processors by increasing the frequency and the number of transistors in the chip, allowing to build faster and more complex single-core processors that could exploit high Instruction Level Parallelism (ILP) out of sequential programs. As shown in Figure 1.1, this trend continued until the early 2000s, when this tecnology scaling enabled rapid growing has encountered two fundamental obstacles: the ever increasing latency of memory accesses due to the speed gap between the processor and the main memory, known as the Memory Wall [84], and the ever increasing power consumption of chips with higher number of transistors and clock rates, known as the Power Wall [58], or the end of Dennard's scaling [34].

In the early 2000s, to overcome the stagnation of single-core processors' performance, the landscape of microprocessors design entered the multicore and many-core era. Shared memory multiprocessors are the most genuine representatives of chip multiprocessors. This family embraces a wide number of chips, from the first commercial multicore processors such as the IBM POWER4, the Intel Core Duo or the AMD Opteron, to current high-end many-core architectures for HPC such as the Intel Xeon Phi or the IBM Blue Gene/Q. The distinctive characteristic of shared memory multiprocessors is its memory organization, composed by a hierarchy of caches with a cache coherence protocol. This scheme allows the different cores to share data without any intervention from the programmer.

Multicore processors can potentially provide the desired performance gains by exploiting the Task Level Parallelism (TLP) of parallel programs, but they have introduced significant challenges for adapting from sequential to parallel computing such as detecting parallel workloads/regions/tasks, distributing and synchronizing those tasks between

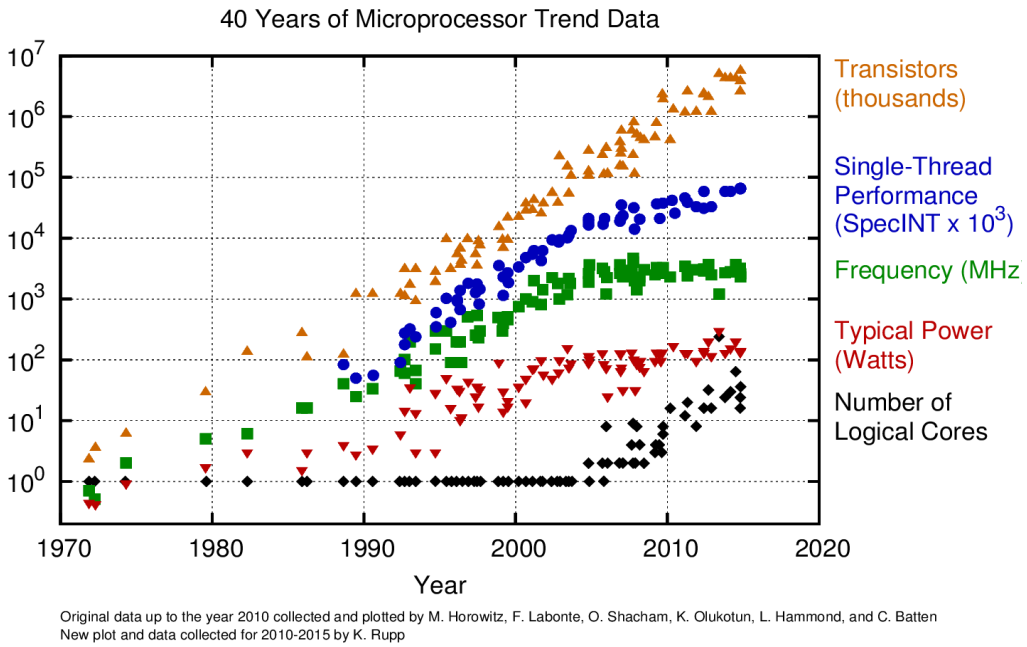


Figure 1.1: Evolution of microprocessors

the available cores. With the trend of ever-increasing core counts in many-core platforms, parallel computing exposes great challenges and responsibilities to the programmer, as to partition the data, evenly distribute different workloads between a large number of cores and communicate between them, which, in the end, significantly degrades programmability.

To further complicate things, computer architecture is exhibiting a tendency towards more heterogeneity, which is promised to be highly effective and power efficient. Some of the important examples are Cell B.E. [51], the SARC architecture [63], the Runnemede [21], the Imagine [53], General-purpose processors plus GPGPUs architectures, Xilinx UltraScale+ MPSoC series [86], Intel Stratix-10 SoC Chips [48], IBM FAbRIC POWER8+CAPI system. All the attention on heterogeneous platforms reflect their potential to offer higher performance and lower energy consumption than traditional multi-core and many-core systems. They can exploit high TLP of parallel programs due to their ability to execute both SMP tasks (tasks that can be executed in any of the cores of general-purpose processors) and heterogeneous tasks (tasks that can be accelerated in specialized hardware units). Unfortunately, they raise new problems such as managing different types of hardware engines and hybrid memory systems during parallel computing. Once more, programmers have to be very aware of the underlying hardware architecture in order to achieve good performance. Moreover, they are also responsible for explicitly transfer-

ing data between memory spaces and handling potential data replications. Additionally, applications often need to be modified to do a good porting to different platforms.

To automate the parallelization and synchronization of workloads in applications and to ease the effort for efficient data movements in heterogeneous systems, task-based programming models are quickly evolving to tackle all these challenges. Significant examples are OpenMP 4.5 [2], OmpSs [35], Codelets [3], StarPU [69], Intel's TBB [46]. Using task-based programming models, an application can be expressed as a collection of tasks with dependences, which are then managed at runtime. They are very simple to use: for example, with OpenMP 4.5 and OmpSs, simply by annotating functions with their data dependences and their types (input, output, inout)[27], the sequential version of application can be transformed into a functional parallel version according to the dependences expressed by the programmer. When the corresponding compilers encounter these annotations, they generate runtime API calls in order to create and submit tasks. Then, at execution time, the runtime will dynamically manage inter-task dependences and schedule tasks for out-of-order execution. They are also very powerful to be able to obtain high performance with applications. An additional important benefit of these programming models is that they decouple the application from the architecture which allows to take advantage of the available information in the runtime system to drive optimization in a generic and application-agnostic way [80, 56, 57]. Moreover they delegate the runtime system the responsibility to exploit task annotations to map the data specified in the task dependences to different memory spaces and schedule tasks to the corresponding execution unit, contributing to the programmability advantages for complex heterogeneous architectures.

However, there are certain non-trivial disadvantages of the default software-only runtimes currently employed. They often have too large overheads that prevent them from exploiting fine-grained task parallelism and heterogeneity. Examining the current existing computer architectures and their future trend, fine-grained parallelism is preferable in a large range of domains for computing. Theoretically with fine-grained tasks, there is a much higher parallelism to be exploited and it is easier to manage workload balance in the system, resulting in higher performance and energy efficiency. In reality the software-only runtime overheads including task dependence management and heterogeneous task scheduling cause performance degradation with fine-grained tasks. Task dependence management overheads are usually related with the way that software runtimes are designed in order to perform their work. For example inserting a task into a task-

dependency graph (TDG) requires the comparison of each of its dependences against those of the existing tasks; in addition, to maintain a global order of tasks, there are often locks to allow only one thread to update the TDG at a time. All of these processes are costly and can incur a lot of thread contention [30, 71, 87, 14]. The problem with heterogeneous task scheduling is closely related to the underlying hardware that applications run. For example, current heterogeneous systems often are consisting of architectures such as big-little cores and general-purpose processors with specialized hardware units. They often have complex memory system such as a mixed layer of shared coherent memory and private non-coherent local memories. Therefore, several threads are often dedicated for those purposes in software-only runtimes because automatically managing task scheduling and data movements among those can be a nightmare. Those management operations are not trivial and they definitely under exploit those threads.

To overcome these deficiencies and extend the benefit of task-based programming models into a finer-grained parallel and heterogeneous computing, in this thesis we propose the Picos system. It is a hybrid hardware/software co-design runtime, which not only reduces critical-path runtime overheads as both the task dependence management and heterogeneous task scheduling are in hardware, but also keeps the flexibility of other software runtime functions.

1.1 Main Objectives, Ideas and Contributions

1.1.1 Main Objective

The main objective of this thesis is to propose a general-purpose hardware accelerated runtime for task-dependence management and heterogeneous task scheduling. It aims to speedup the runtime system, meanwhile reducing the contention among all the thread executing runtime functions. It should be easily adaptable to be integrated with the runtimes of different task-based programming models, transparent to application programmers and even to the programming model that it accelerates. In addition, it should also be easily adaptable to manage a large variety of hardware units like SMP, Big-little cores, FPGA, ASIC, GPU, ..., etc. Finally, as one of the most concerned features in parallel computing, it should be energy efficient.

1.1.2 Main Ideas

To achieve our objective, we describe three main ideas that have driven this research work.

Task dependence management.

With task-based programming models, an application can be expressed as a collection of tasks with data dependences of type input, output or inout (both input and output), and the runtime system analyzes the inter-task dependences and ensures the correct execution order of the program. One critical runtime overhead is constructing and managing the TDG during the application execution. For this reason, we propose a hardware architecture (Picos) to accelerate this part and be seamlessly integrated with the software runtime. The guidance for the functionality of this architecture should be as follows: when a new task is created in the software side of the runtime, Picos should read the task and its dependences, and insert it into a TDG; when all the dependences of a task in the TDG are ready, Picos deems the task as ready and schedules it to execute in threads; when a task finishes executing in a thread, Picos should read this information and finalize its remaining role. If this finished task has successors that depend on it, then these tasks are awaken; afterwards this task is deleted from the TDG directly. The implementation of the design should focus on balancing the speed and cost among many other trade-offs.

Nested task support.

With the support for task dependence management as described above, we are able to handle most of the regular and irregular application dependence patterns. However, to be a truly general-purpose hardware task-dependence manager, nested task support is a necessary feature. Nested tasks are those that have been created by another task. They are often seen in applications with recursive algorithms or programmed with libraries that embedded tasks inside. However, to the best of our knowledge, there is no nested task support in the State-of-the-Art hardware task-dependence managers. The reason for that is due to the limited resources of hardware-based implementations after being implemented or tapped, which can lead to a system deadlock. For this, we propose a novel hardware/software co-design, based on a deadlock-free architecture, that includes nested task support.

Heterogeneous task scheduling.

Heterogeneous architectures are ubiquitous in present days, for they are able to provide higher performance and energy efficiency. One of the challenges for managing them is the task scheduling and data movements among their often-complex memory systems.

With a software-only runtime of task-based programming models, the cost for launching computations and managing data movements is high. Since Picos has already managed all the tasks and their dependences inside hardware, it has all the information required to manage task scheduling and data copies for heterogeneous hardware execution units. Therefore, we extend Picos with the followings.

1 Very tied co-processors or accelerators directly managed by Picos.

2 Heterogeneous task scheduling that optimizes the execution of tasks.

With these three main ideas, during our research, we have built three prototypes that tackle each of these ideas one at a time, with each prototype being built on top of the previous one. Besides these implementations, we have also spent a lot of time on the integration process in order to be able to obtain real measures that can compare against current software runtimes. We categorize this process into three different integrations.

1 Hardware integration. By this, we refer to the connection of Picos hardware task dependence manager and scheduler with the SMP in a SoC chip. Later, when we test the heterogeneous task scheduling, we also need to connect numerous HW functional accelerators with Picos and SMP in a MPSoC chip.

2 Runtime or programming model integration. It refers to the process that we isolate the TDG management and task scheduling in the software-only runtime, and replace them with the hardware manager and scheduler. This is partly achieved by developing the Picos API libraries.

3 OS integration. It refers to modify the fsbl, boot files and devicetrees to ensure that Linux recognizes and is aware of the hardware platform that includes our custom designs.

1.1.3 Main Contributions

The main contributions of our research can be summarized as follows:

- A high speed and energy efficient hardware task dependence manager. It accepts newly created tasks and their dependences, constructs the TDG in hardware and schedules those tasks to be executed, when they are ready, in any of idle threads running in the SMP.

- A novel hardware/software co-design supporting nested tasks. It is developed to prevent potential system deadlocks when the hardware resources are fully used or when internal memory conflicts appear in the Picos task dependence manager, thus allowing the design to be more general purpose. To the best of our knowledge, this is the very first time that a nested-task feature has been included, described and functionally implemented in a hardware task dependence manager.

- A new heterogeneous task scheduling support in hardware. In a real system including SMP and numerous other hardware execution units, tasks are scheduled to a suitable destination and data movements are automatically managed to shorten the total execution time of the application. This is realized by constantly tracking the workloads waiting for each hardware component in the system, and scheduling tasks to the component that has the least amount of waiting jobs and has the highest priority. It does not require any profiling and the workloads are well balanced with fine-grained tasks.

- A fully integrated and functional hybrid hardware/software runtime based on real commodity hardware platforms. A State-of-the-Art task-based programming model OmpSs is up and running with Ubuntu Linux, with our hybrid runtime. The base hardware platforms are Xilinx Zynq-7000 series SoCs and Ultrascale+ MPSoCs, which include the SMP threads, the Picos task-dependence manager and heterogeneous task scheduler, and numerous HW functional accelerators (HwAccs, used as proof-of-concept heterogeneous hardware execution units).

- Detailed study of scalability and energy consumption for each Picos prototype, with both synthetic and real applications. For the first and second prototype, beside all the task-dependences being managed in Picos, all the tasks are scheduled to execute in SMP. For the third prototype, all the tasks are scheduled to execute in both SMP and HwAccs. All the performance and energy consumption results are compared with a cutting-edge parallel task-based programming model.

- Visualization analysis of real application executions on real hardware, with tasks executed in both threads and HwAccs. This gives insight on application execution, Picos activities on both homogeneous and heterogeneous systems. This insight also opens future directions on higher energy saving designs.

During the development process, there are other practical and more general gains that

are worth mentioning. First, it offers insight on how to design from scratch a hardware architecture and different layers of integration with other hardware, with software runtime, and with high level OS such as Linux. Second, we realize that many hardware execution units are kept busy-waiting when there is not parallelism to exploit. This leads to a huge waste of energy. Thus, a future feature can be to use hardware to sleep/wake-up them to achieve a higher energy efficient system without influencing performance. Finally, from our experience, we conclude that the most beneficial design is a combination of the software and hardware, as they both have their limitations and advantages. Indeed, with each new advancement of technology, algorithms that were previous more suitable in software can become more beneficial if implement in hardware, and vice-versa. Therefore, the proper software/hardware combination will contribute to achieving a performance and energy wise architecture.

1.2 Publications

In this section, we briefly summarize the publications derived from this research.

1 **"Task dependences management hardware acceleration for task-based dataflow programming models" [77]**, in *Proceedings of the 3rd International BSC Doctoral Symposium*. 2015. 1st author. This paper describes the preliminary research and results of our proposals.

2 **"Performance Analysis of a Hardware Accelerator of Dependence Management for Task-based Dataflow Programming models" [71]**, in *ISPASS '16: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2016. 1st author. This paper describes our very first prototype Picos. It is a major milestone as it built up the foundation architecture of our research. It is a hardware task-dependence manager designed in VHDL, implemented and tested in a Xilinx Zynq-7000 series SoC platform. This work has also been published and presented in other places [76, 4].

3 **"Picos, A Hardware Task-Dependence Manager for Task-Based Dataflow Programming Models" [73]**, in *HPCS '17: Proceedings of the International Conference on the High Performance Computing and Simulation*, 2017. 1st author. This paper decribes the preliminary research and results of our second prototype Picos++.

4 **"General Purpose Task-Dependence Management Hardware for Task-based Dataflow Programming Models" [72]**, in *IPDPS '17: Proceedings of the 31th IEEE International Parallel and Distributed Processing Symposium*, 2017. 1st author. This paper describes our second prototype Picos++, which introduced a new features for nested task support, allowing it to be a general-purpose hardware task-dependence manager. It was designed in VHDL, implemented and tested in the same SoC platform as our work. It is a fully operational system with integration in hardware, runtime and Linux. Results of performance and energy consumption are obtained during real application executed on real hardware system.

5 **"Characterizing and Improving the Performance of Many-Core Task-Based Parallel Programming Runtimes" [14]**, in *IPDPS '17: Proceedings of the 31th IEEE International Parallel and Distributed Processing Symposium Workshops*, 2017. 2nd author. This paper introduces a full implementation of a centralized runtime manager DAST with automatic load balancing between the manager and the workers. As mentioned earlier, the runtime integration requires the isolation of task-dependence analysis from software and support for Picos APIs. The centralized software runtime described inside has been used to integrate with the Picos prototypes.

6 **"Hardware Heterogeneous Task Scheduling for Task-based Programming Models" [74]**, in *The 2018 OpenMP Developers Conference (OpenMPCon)*, 2018 (accepted). 1st author. This paper mainly focuses on describing the runtime support for heterogeneous task scheduling in the third prototype of Picos++, for task-based programming models. Preliminary results with real applications are presented and analyzed.

7 **"Asynchronous Task Creation for Task-Based Parallel Programming Runtimes" [16]**, in *The 2018 OpenMP Developers Conference (OpenMPCon)*, 2018 (accepted). 2nd author. Although the runtime systems of task-based programming models have the ability to manage different hardware accelerators, they employ a master/slave model where only the threads can create tasks and the hardware accelerators are only consumers, which eventually limits the system possibilities. This paper proposes a general organization to allow the accelerators to interact with the runtime in order to create tasks and synchronize with them.

8 **"Picos++: a Hardware Accelerated Runtime for Task-dependence Management and Heterogeneous Task Scheduling" [75]**, in *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2018 (submitted for review). 1st author. This paper describes the third prototype, which introduced a new feature to support heterogeneous task scheduling and to automatically manage data movements among different memories in hardware. It is a fully operational system in real hardware in a Xilinx Ultrascale+ MPSoC platform. Results of performance and energy consumption are obtained for real benchmarks executing on the Picos++ system with up to 4 SMP threads and 15 HW functional accelerators. It has also been shown in the RoMoL final workshop [5].

9 **"Application Acceleration on FPGAs with OmPss@FPGA" [15]**, in *the 2018 International Conference on Field-Programmable Technology (FPT)* (accepted). 2nd author. This paper presents the OmpSs@FPGA toolchain, which extended the OmpSs programming model with the support for automatically generating and mapping HW functional accelerators in FPGA for functions annotated by specific pragmas. Real applications such as Matrix Multiplication, Cholesky and N-Body are used to evaluation this toolchain on a Zynq Ultrascale+ MPSoC.

1.3 Thesis Structure

The contents of this thesis are organized as follows:

Chapter 2 first provides a general review of several task-based programming models in literature, focusing on OpenMP and OmpSs. Then it remarks some of the most representative architecture works that are related to this thesis.

Chapter 3 first introduces hardware infrastructure, focusing on how Picos works and connects with other hardware execution units in the system. Then it shows the two runtime systems used to evaluate the system mentioned. Afterwards it shows both the synthetic and real benchmarks selected for the functionality and performance examination. Finally, it presents the metrics for testing and evaluating our prototypes.

Chapter 4 presents our very first hardware prototype of Picos, which manages all the task-dependences in hardware. To better explain the context, this chapter starts with the introduction of the existing problems in the default software-only runtimes of task-based programming models; followed by a top-to-bottom explanation of organization of the first prototype Picos. Afterwards, we discuss experimental setup and benchmarks,

present detailed performance and scalability studies of Picos, and analyze the results.

Chapter 5 describes a hardware/software co-design to support nested/multi-level tasks with dependences. It extends the benefit of the task dependence management in the Picos++ system to a much larger set of applications. For applications with recursive algorithms or libraries embedded with tasks, the nested task support greatly simplifies the programming effort and improves nested parallelism. In this chapter, we also present and analyze results of performance and energy consumption.

Chapter 6 proposes a heterogeneous task scheduling policy to effectively schedule ready tasks to suitable hardware execution units with the least amount of active waiting work and with the highest priority in the system. It also manages the corresponding data movements among the shared memory and local memories. As a proof-of-concept implementation, we have constructed systems including SMP, Picos++ and numerous specialized HW functional accelerators for task execution.

Chapter 7 concludes this dissertation by remarking its main contributions and by providing a brief summary of the future work.

Chapter 2
State of the Art

This chapter first gives an overview of the State-of-the-Art task-based programming models, focuses on OpenMP and OmpSs. Then, we take a look of numerous representative related works on hardware task-dependence management and heterogeneous task scheduling in the literature.

2.1 Task-based Programming Models

Many task-based programming models have emerged in recent years to face the expected complexity of future multicore, many-core and heterogeneous architectures. These programming models conceive the execution of a parallel program as a set of tasks with dependences among them.

In task-based programming models the programmer only has to express or define the tasks and their dependences in the sequential code. With this information, the runtime system manages the parallel execution of the tasks, taking care of scheduling tasks to different cores and synchronizing them without any intervention from the programmer. In order to manage the execution of the tasks the runtime system constructs a TDG, which is a directed acyclic graph where the nodes are tasks and the edges are dependences between them. Similarly to how an out-of-order processor schedules instructions, the runtime system schedules a task on a core when all its dependences are satisfied, and when the execution of the task finishes, its output dependences become ready for the successor tasks.

OpenMP 3.0 [8] provides compiler directives to support basic tasking constructs, that are extended with data dependences in OpenMP 4.0 [59], while OmpSs [36] extends OpenMP 4.0 with additional directives to specify task priorities and special tasking constructs. The latest release of OpenMP 4.5 [2] and OmpSs [27] further provide `pragmas`

to specify the types and number of instances of available hardware devices in the underlying system. These `pragmas` are also used to assist heterogeneous task (task that can be executed in heterogeneous hardware units) scheduling and its coupled data movements between different memories.

The Codelet model [90, 3] breaks applications into codelets (a similar concept as tasks) with data or control dependences. A codelet is a (usually short) sequence of machine instructions that executes until completion. Once it is scheduled to execute, it cannot be interrupted and migrated elsewhere. Each codelet is associated with a locale, which gives a high-level description of available hardware components as a suggestion to where it can be scheduled to execute. The Codelet model uses Codelets Graphs (CDG) to manage the execution orders. When all the data or control dependences of a codelet instance are ready, it can be scheduled to execute. When a codelet finishes, it releases some resources or produces some data items. The other codelets that are dependent on this finished one will then becomes ready and be scheduled.

StarPU [6, 69] is a runtime system that offers support for heterogeneous multicore architectures. It supports task-based programming models. Applications submit computational tasks, with CPU/GPU implementations, and StarPU schedules these tasks and manages associated data transfers automatically between accelerators and the main memory transparently. One of the important data structures in StarPU is task, by default task dependencies are inferred from data dependency and they are managed by the runtime in StarPU.

Others such as Sequoia [39] uses tasks to express the data movements among abstract complex hierarchical memories that existed in the system. Each Sequoia task is an explicit expression of data movement through the memory hierarchy. Tasks have their private address space and the programmer organizes them hierarchically. Legion [10] programs are decomposed in tasks that access data partitions and potential execution hardware components manually specified by the programmer. Charm++ [52] is a C++ based asynchronous message driven programming model where the programmer decomposes a program into message-driven objects called chares. Chares are distributed among the processors and the runtime system is in charge of sending messages to the chares to trigger the execution of the code within them.

Intel TBB [64] is a C++ template library that implements a task execution model where the programmer splits the serial code into tasks that have implicit control dependences with their parents and child tasks. The latest release Intel TBB 4.4 introduces a

flow graph feature with enhancements to specify concurrency, external communications, and a composability layer to support heterogeneous computing.

Cilk [12] extends C and C++ with keywords to spawn and synchronize tasks, and uses a simple fork-join model enhanced with work-stealing primitives to balance the load efficiently. Vandierendonck et al. [82] proposed to extend Cilk with data dependences between tasks.

The characteristics of task-based programming models make them very suitable for current and future homogeneous and heterogeneous architectures. The main reason is that these models allow the programmer to specify parallelism in an architecture-agnostic way [80, 18, 60, 56, 57, 40], hiding from the programmer the complex architectural details and enabling the same code to be executed in very different platforms. In addition,the data dependences of task-based data-flow programming models can be used to program not only shared memory multiprocessors but also heterogeneous architectures that require explicit data transfer between address spaces.

2.1.1 OpenMP

OpenMP is the most commonly used programming model for shared memory multiprocessors. It allows to specify parallel constructs in C, C++ and FORTRAN using simple and portable compiler directives. These directives are supported by the vast majority of modern compilers and operating systems for shared memory multiprocessors. The core elements of OpenMP are the directives to specify parallel regions, workload distribution, data-environment management and thread synchronization.

OpenMP 3.0 uses a fork-join parallel execution model, where a single thread is used in sequential regions and the execution branches off in parallel at designated points in the program, specified by the programmer with the `#pragma omp parallel` directive. OpenMP also allows to specify how the work is distributed between threads in a parallel region, either assigning independent blocks of code to each thread using the `#pragma omp section` directive or distributing the loop iterations among threads using the `#pragma omp for` directive. Loop parallelism is the prominent feature offered by OpenMP 3.0.

Besides a lot of features provide for loop parallelism, OpenMP 4.0 officially introduces the task dependences into the programming model. By using `#pragma omp depend(dependence-type: list)`, where the dependence-type is one of the following as input (in), output (out), input and output (inout), applications can be ex-

pressed as a collection of tasks with dependences. Nested task support requires synchronization between parent and children tasks, this synchronization can be realized either by implicit task dependences or by explicitly using taskwait. It also provides a set of directives to instruct the compiler and runtime to offload a block of code to the device. Such as `#pragma omp target map(to: variable, array-lists) map(from: variable, array-lists)`, by combining with the depend clauses, they can be used to schedule task and map variables/move data between the host and target devices, which can be SMP, GPU, FPGA etc. During the execution of tasks in target devices, the host device has to start and wait for the execution.

OpenMP 4.5 introduces major features for task support. Task priority is added by using `#pragma omp task priority(priority-value)` to support hints that specify the relative execution priority of explicit tasks. Taskloop constructs are added to support specially nestable parallel loops. It also continues the effort to add more features and clarifications of the device constructs for offloading tasks and data mapping to different devices. For example, by combining depend clauses and `#pragma omp target nowait`, with `map(to: variable, array-lists)`, further with `map(from: variable, array-lists)`, this allows the host thread to perform other work while asynchronously waiting for the target region execution to complete.

The directives introduced by the programmer are processed by the compiler to generate parallel code. In this process the compiler arranges the code so that the parallel regions are encapsulated in separate functions/tasks, it sets up the declaration of the variables according to its sharing attributes, and it adds function calls to the runtime system in the points of the code where the parallelism is created, executed and synchronized. When the code is executed the runtime system is in charge of managing the threads, tasks and devices. For this purpose the runtime system provides routines to create threads, synchronize them, and destroy them. In order to assign tasks to threads the runtime system implements schedulers and work queues that support all the forms of parallelism allowed by the OpenMP directives.

This coordinated effort between the compiler and the runtime system is what allows OpenMP and also OmpSs in the following to generate parallel code and to manage the parallel execution from simple directives. This model has been very successful because it allows to exploit the capabilities of shared memory multiprocessors and heterogeneous systems with a programming interface that is easy to use for programmers and portable across many architectures and systems.

2.1.2 OmpSs

The OmpSs programming model [27], developed by the Programming Models group at Barcelona Supercomputing Center (BSC), is an effort to integrate different features from the OpenMP standard and the StarSs programming model family. The name of OmpSs is a combination of these two programming models.

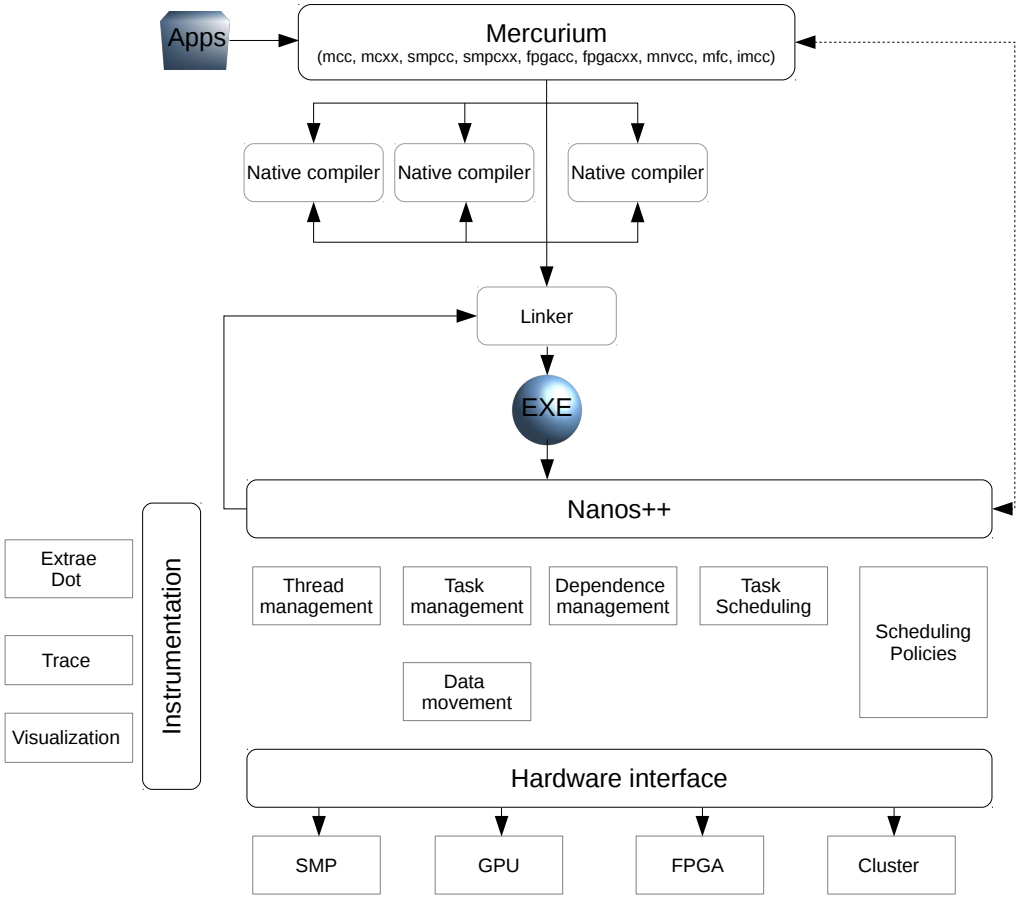


Figure 2.1: OmpSs operational flow

OmpSs inherits from OpenMP 3.0 the philosophy to develop parallel code: begin from a sequential program and annotate candidate regions or functions in the source code with certain directives, to guide the compiler on the transformation into a parallel program. The difference and beneficial feature is that, instead of using a fork-join model as in OpenMP 3.0, OmpSs uses a thread-pool model. By using OmpSs pragmas, such as `#pragma omp task in(variable or array-lists), out(), inout()`, applications are abstracted as a collection of tasks, and the runtime tracks the inter-task dependences and copies them into a ready task pool when all their predecessors are ready. Threads are

free to acquire ready tasks for execution when they are idle. This approach is specially beneficial when trying to accommodate irregular applications. For this reason, OmpSs strongly influenced the OpenMP 4.0 releases [8].

Similarly, the nested task support can be realized either by implicit task dependences or by explicitly using `#pragma omp taskwait`. Additionally, OmpSs task directive allows the programmer to specify data dependences and to map the execution of certain tasks to different types of hardware devices. By using `#pragma omp target device(device-type-list)`, tasks of applications can be scheduled to execute in different type of devices including SMP, FPGA, GPU, ..., etc. Optional clauses, for example `onto(accelerator-id)` and `num_instances(n)`, can be appending to the previous directive when FPGA is used, to decribe the types and number of accelerators that should be created and can be used at runtime. Optional clause like `(copy_deps)` can be used to inform the runtime to also copy the data specified by the data dependences between the host processor and the target devices.

OmpSs is composed of the Mercurium source-to-source compiler [9] and the Nanos++ runtime. As can be seen in Figure 2.1 [13], an OmpSs application is first compiled by Mercurium which applies the source-to-source transformations and replaces annotations by API calls to the Nanos++ runtime library. After that, the native compiler is used to generate the object files, which are linked against the Nanos++ library to generate the executable. When the application is executed in the system, the Nanos++ runtime creates one Work Descriptor (WD) for each task. Each WD includes all the information required to mange the task through its life time. It includes information such as task id (including parent and child task id), all the memory addresses and directions of its dependences, etc. The parent task contains the task-dependency graph of its children, this ensures that only tasks from the same parent are dependent on each other (one of the definition in OmpSs and OpenMP standard). In this way, the global order can be managed by a set of distributed and hierarchical TDGs and is guaranteed because a parent always has a super-set of all its children dependences.

An OmpSs task typically has six stages in the runtime. (1) Task creation, when the WD is allocated and initialized; (2) Task submission, when the dependences of this task are submitted to the task-dependency graph, the runtime computes its relationship with others that arrived earlier and inserts it at the right place. (3) Task ready, when all its dependences are resolved, the runtime schedules it to an idle thread. Otherwise it is (4) task blocked. (5) Task finalization, when a task finishes execution, runtime checks whether

there are any successors of this task and updates the readiness of their dependences. Finally (6) when there are no more successors, its WD is deleted by the runtime.

Our proposal is to use hardware to accelerate the task-dependence management and heterogeneous task scheduling, therefore mainly the `Dependence Management`, `Task Scheduling` and `Data movement` part in Figure 2.1 will be served by the hardware.

With the pragmas for different target devices both supported in OpenMP 4.5 and OmpSs, there are different options for the communication between the host processors and heterogeneous devices such as FPGA in this case.

OmpSs@FPGA [83] proposes different ways to communicate between processors and FPGA. One of the proposals is to use FPGA as a master module, in which commands (including tasks and dependence addresses) are sent to FPGA by the processors and all the data movements are managed by the accelerators themselves.

OpenCL implementations [31, 47] use similar approaches for scheduling workloads to FPGA as it uses for GPUs. In this case, there are four different types of memories that the runtime may have to manage in the task execution. Host memory connects to the host processor directly, global memory is shared between processor and FPGA, while local and private memories are inside the FPGA. Data is stored in the host memory by default, to schedule workloads and transfer data to the FPGA, the programmer has to allocate space in global memory and issues write/read commands to copy data between host and global memories, and from local memory to global memory. Different FPGA accelerators or kernel instances can only communicate through the shared global memory. In OpenCL, the data communication has to be 64-byte aligned, and is initiated by the processor and will be performed using DMA transfers.

OpenARC is an implementation [55] that translates OpenACC applications into OpenC, therefore it inherits the same communication approaches as described above with OpenCL implementations. The difference is that it introduced two new features: kernel-pipelining and dynamic memory-transfer alignment. The first one allows different FPGA accelerators or kernels to communicate through the FIFO channels between them, this is faster than to go through the global memory every time. The second one overcomes situations when the data between host and device memories are not fully 64-byte aligned. Despite that, all the communications are performed by DMA transfers and have to be managed by the processor.

OpenMP 4.5 [2] has the standard with similar processor-FPGA communication approaches. There are some works [19, 68], similarly to OpenCL implementations, FPGA

is treated as a passive accelerator, and all the communication is started by the host CPU through DMA engines.

Using FPGA accelerators as slave modules (that rely on processors to perform data movements) allows for simpler designs on the communication mechanism between the processors and the accelerators. However, the overhead is large and only large amount of data movements can be justified with such methodology. On the other hand, using FPGA accelerators as master modules that can read/write memories on their own greatly reduces the amount of works required from the processors, and ultimately improves the data movement speed. In our proposal, the HW accelerators use a master mode for the communication with the processors.

2.2 Related Hardware Proposals

There are several research works focusing on similar topics as addressed in this thesis. This section shows some of the representative ones in the literature that have inspired us on our research work. Some of those works focus on hardware support for task dependence management and task scheduling. However, the programmer had to manage the inter-task dependences. They were either studied with gem5 [11] or other simulators, or were synthesized or even implemented in real hardware. Intel CARBON [54], Asynchronous Direct Messages (ADM) [65] and Task Scheduling Unit [43] introduced hierarchy hardware queue architectures to speedup task stealing and scheduling. Although these works did not include task dependence analysis, their hierarchy hardware queues are a really general-purpose and useful way for fast task scheduling, which can be used to schedule tasks both to homogeneous or heterogeneous systems.

Video-oriented task scheduler [1] and Programmable Task Management Unit (TMU) [67] were inspired by the works mentioned earlier. Besides with hierarchy hardware queues, they added hardware support to accelerate task creation, synchronization and scheduling for applications with specific dependence patterns found in video processing domains. Multilevel Computing Architecture (MLCA) [20] introduces a novel multi-core architecture for coarse-grained task parallelism for multimedia applications. The MLCA augments a traditional multicore architecture to serve as low level processing units (PU) with a high level control processor (CP). The CP employs task queue, register renaming, out-of-order execution to dynamically detect the inter-task dependences and schedule tasks when they are ready. Although these works are specialized for video/multimedia applica-

tions and coarse-grained parallelism, they share the common belief that hardware can be used to accelerate task-dependence analysis and task scheduling more efficiently.

Swarm [50] uses the co-design of the execution model and micro-architecture to exploit ordered irregular parallelism in task-based parallel applications. It relies on speculative task execution and conflict detection to preserve dependences. Swarm requires hardware support for speculation rather than for dependence management and uses either a FIFO queue or a spatial scheduler fixed in the architecture [49]. Fractal [70] extends Swarm to allow nested parallelism by means of task domains, that can be ordered or un-ordered to avoid over-serialization.

F. Yazdanpanaha [89] proposed a task dependence manager architecture for task-based dataflow programming models for fine-grained parallelism based on simulation results. It inputs newly created tasks and their data dependences, constructs task-dependency graph and schedules ready tasks dynamically to idle threads. E.Castillo [24] proposed a similar hardware task-dependence manager, studied with gem5 simulator for design space exploration. Both research works proved that using hardware for task-dependence management can be beneficial for fine-grained parallelism. The difference is that the first work also believes in using hardware for task scheduling while the second one shows preference in using software for task scheduling. In our opinion, the combination of both software and hardware scheduling is beneficial depending on the hardware platform used.

Nexus# [32], Task Superscalar [88] were also proposed to accelerate task dependence management in hardware for task-based dataflow programming models. They were both coded in industrial standard hardware design language Verilog and VHDL. Task Superscalar was discontinued due to some design deadlocks, and inspired Picos as an early example of a Runtime-Aware architecture [81, 23]. On the other hand, Nexus# was evaluated using traces of real applications. The performance results proved to be better than using software alternatives. Our proposals have several new contributions in addition to improve performance over Nexus#. They require less than half of the hardware cost, add nested task support, allow the execution of a much larger range and general-purpose applications, and finally also include heterogeneous execution support.

Task scheduling has been studied intensively and there are a large number of published paper works. In the following, we present some of the representative ones that could be combined with our proposals.

The Heterogeneous Earliest Finish Time (HEFT) algorithm [78] maintains a list of tasks sorted in decreasing order of their upward rank. At each schedule step, HEFT as-

signs the task with the highest upward rank to the processors that finishes the execution of the task at the earliest possible time. The Critical-Path-on-a-Processor (CPOP) algorithm [78] also maintains a list of tasks sorted in decreasing order as in HEFT, but in this case it is ordered according to the addition of their upward rank + downward rank belong to the critical-path. Both of them are static algorithms.

Kallia Chronaki [28, 29] proposed a criticality-aware task scheduler (CATS) and a critical-path scheduler (CPATH), to dynamically assign critical tasks to fast cores in a heterogeneous multicore and non-critical tasks to the slower cores in the system. The first method considers critical tasks as tasks that are in a longer dependence chain; the second approach is based on the first one, in addition with the consideration of task execution time. Both of them prioritizing the newly-created tasks at runtime and updating the criticality of tasks in the dynamic task dependency graph.

Judit Planas [62] described a scheduling policy where different implementations of functions suitable for different hardware units are managed dynamically. Their policy has two stages, first a training stage where the scheduler learns about the execution time of each implementations in the system. On the second stage it tries to schedule the task to the hardware unit that has an estimated earliest finish time. In our last proposal, we include this concept for heterogeneous executions implying hardware accelerators.

2.3 Heterogeneous systems

There exist many heterogenous architectures. Some of the main representative examples, types of accelerators used on those architectures, and software support for those architectures are described in this section.

The Cell B.E. [51] is one of the first breakthrough heterogeneous multicore processors for HPC and multimedia workloads. The architecture consists of a general-purpose core called Power Processor Element (PPE) and eight accelerator cores called Synergistic Processor Elements (SPEs) connected through a high bandwidth NoC named Element Interconnect Bus (EIB). PPEs have shared memory with a global address space, while SPEs have their own private virtual and physical address spaces and are not coherent with the rest of memories in the architecture. Each SPE can issue load and store instructions only to its private memory and a DMA control unit is used to transfer data from or to the rest of memories in the architecture and maintains the coherence with PPE. The SARC architecture [63] is a heterogeneous architecture with clusters of master and worker cores.

Similarly to the Cell B.E., the master cores are general-purpose cores responsible for running the OS, starting applications and spawning tasks to the worker cores. The worker cores in charge of running the tasks have different ISAs and pipelines optimized for different applications. The Runnemede [21] is a modular and hierarchical architecture. The basic module of the Runnemede is called block, which consists of a general-purpose core, eight execution engines, an intra-block network. The Imagine [53] is a stream accelerator for media processing. Its architecture consists of a microcontroller that stores VLIW instructions, eight arithmetic clusters with eight functional units each, a local register file, and a 128 KB Stream Register File (SRF) as an array of memory. The Merrimac [33] is a stream accelerator with a very similar architecture. It has twice the number of arithmetic clusters with a different mix of functional units that are better suited for HPC workloads, a correspondingly larger SRF.

General-Purpose Graphics Processing Units (GPGPUs) are accelerators designed to execute massively parallel workloads very efficiently. These architectures have received a lot of attention during the last years, specially the ones manufactured by NVIDIA, Intel and AMD. GPGPUs are often attached to a general-purpose processor (called the host processor), and they have different address spaces[1]. The host processor is in charge of running the OS, starting applications and executing the sequential parts of the applications. The GPGPU executes those parts of the applications that expose a high degree of parallelism, which are encapsulated in kernels. When the application encounters a kernel, the host processor transfers the code of the kernel and the data accessed by the kernel to the GPGPU, triggers its execution, and transfers the data back to the main memory if needed.

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. They can be programmed to desired applications or functionality requirements after manufacturing. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks. FPGAs especially from Xilinx and Intel (formerly Altera) are widely used in many different domains like Aerospace and Defense, Media, Automotives, Data Center, Medical and High Performance Computing.

[1] Architectures consist of host processors and GPUs usually have separate physical addresses, at least for discrete GPUs with their own physical memory. Integrated GPUs often shared physical memory but had no efficient way to access CPU's memory space, but with latest ones (especially from AMD) supporting virtualization and address translation for virtual memory, the border has become blurred.

Multiprocessors plus FPGAs are revolutionary hardware architectures that are getting more popular and successful in recent years. They are able to provide high performance at a low energy cost as the other heterogeneous architectures mentioned earlier. They are also flexible to allow reconstructing different heterogeneous system through reprogramming, with a much faster developing period and a much lower cost. Top hardware and software vendors have started making it a standard to incorporate FPGAs into their compute platforms for performance and power benefits. For example, Xilinx Zynq UltraScale+ MPSoC series [86], which offer up to 4 ARM A53 cores for general-purpose computing, includes FPGA logic for customer functions. Intel's new Altera-powered Stratix-10 SoC Chips [48] have a similar architecture. IBM Coherent Accelerator Processor Interface (CAPI) [44] for POWER8 Systems provides an easy-to-use API to attach custom acceleration engines to the coherent fabric of the POWER8 chip. IBM FAbRIC POWER8+CAPI system is a cluster of several x86 servers and nine POWER8 servers. The x86 nodes serve as the gateway node and build machines for running FPGA tools. Each POWER8 node is a heterogeneous compute platform equipped with three accelerating devices: a Nallatech 385 A7 Stratix V FPGA adapter, an Alpha-data 7V3 Virtex7 Xilinx-based FPGA adapter and a NVIDIA Tesla K40m GPGPU card. FPGA boards are CAPI-enabled to provide coherent shared memory between the processor and accelerators.

Chapter 3
Methodology

This chapter describes the experimental methodology followed in this thesis. The first section describes the development infrastructure, focusing on how Picos works with other hardware devices. In addition, the two runtime systems used to evaluate the system are described. The second section shows both the synthetic and real benchmarks, with their main characteristics and the setup employed in the experiments to execute them. Finally, the third section defines the metrics used to evaluate the proposals of this thesis: task and dependence repetition rate, performance, hardware resource utilization and energy consumption.

3.1 Development Infrastructure

Picos aims at accelerating task dependence management and heterogeneous task scheduling for task-based programming models in order to overcome the software-only runtime overheads for fine-grained parallelism and heterogeneity.

Figure 3.1 shows a simple illustration of how a hardware architecture with Picos inside could look like. The left side of the figure shows how Picos is connected to threads, N different HW functional accelerators and some other types of hardware. Picos receives new tasks from threads and manages all the task dependences; when a task is deemed ready, Picos sends it to be executed in either threads or in other hardware devices; finally, all the finished tasks are sent back to Picos to update its internal task dependency graph. On the right side, one of the hardware platforms selected for prototyping Picos is shown. This hardware platform is called the AXIOM board[1], it includes 4 ARM Cortex-A53 cores and a FPGA for custom functions. When we map the Picos system on top of the platform, the software part of the runtime and the OS operate inside the ARM cores. The

[1]Zynq Ultrascale+ based board designed and developed under AXIOM european project[66].

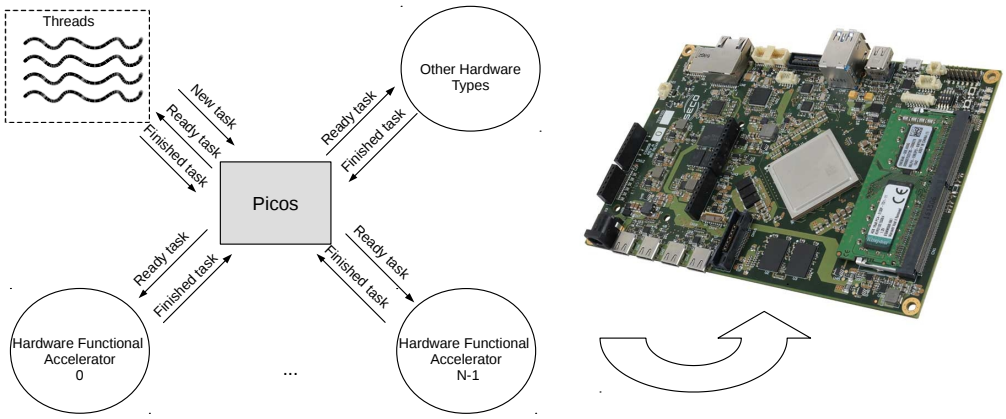


Figure 3.1: A general view of Picos and other hardware devices

Picos hardware, its communication interconnection and the HW functional accelerators reside in the FPGA part.

In the following sections, first the hardware platforms, the design tools and languages selected are described. Then, the HW functional accelerators shown in Figure 3.1 are explained along with their connection to the system; Afterwards, the two runtimes - the baseline software-only and the Picos runtime - operating on top of our system are shown.

3.1.1 Design Tools and Languages

The different prototypes of Picos have been coded in VHDL (VHSIC Hardware Description Language), while the communication logic for Picos and the HW functional accelerators have been coded with C++ with Xilinx High-Level Synthesize (HLS) directives.

VHDL together with Verilog are the two most widely used hardware description languages used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays (FPGAs) and integrated circuits (IC), in both academic and industrial domains.

High-Level Synthesis (HLS) directives is an effort from Xilinx to facilitate the programming for FPGA-based designs. Instead of using low-level HDL languages, programmers can design certain circuits by adding HLS directives into C/C++ source code, allowing them to be transformed into VHDL/Verilog codes. It is convenient and time saving for offloading mathematic functions and managing communication networks between processors and custom designs.

In this thesis, Xilinx Vivado Design Suite 2014.4 was used for developing the first prototype of Picos, version 2015.4 was used for developing the second prototype, and

Table 3.1: Main characteristics of Zedboard and the AXIOM board

Platform		Zedboard	AXIOM
Main Chip		**XC7Z020**	**XCZU9EG-FFVC900**
SMP	Core count	2 ARM Cortex-A9	4 ARM Cortex-A53
	Core Frequency	667MHz	1.1GHz
	L1/L2 Cache	32KB/512KB	32KB/1MB
FPGA	LUTs	53,200	274,080
	FFs	106,400	548,160
	BRAM_36Kb	140	912
	DSP(18x25 MACCs)	220	2,520
	FPGA Frequency	100MHz	50 to 300MHz
Main memory		512MB DDR3	4GB DDR4

version 2016.3 was used to design and synthesize the third prototype of Picos system.

3.1.2 Hardware Platforms

Two main hardware platforms have been used to develop and evaluate different prototypes of Picos. The common characteristic among these platforms is that they all include general-purpose processors where the OS and software runtime can operate, and a FPGA part where Picos and many other hardware accelerators can be configured. In addition, these hardware platforms offer tight integration between the processor and the FPGA through AXI interconnection network, which is very suitable for our user case for fast and small amount of data exchange pattern.

Table 3.1 summarizes the main specifications of these hardware platforms. Each platform contains a main chip and a main memory. Each main chip includes two main parts: SMP and FPGA resources.

The first hardware platform shown is Xilinx Zedboard[7], it has a Zynq-7000 series SoC chip XC7Z020 [85] which includes 2 ARM Cortex-A9 cores (operating at 667MHz) with a shared main memory of 512MB DDR3 and a FPGA. By default, the ARM cores and FPGA are not connected. However, users are free to design the interconnection network and communication schemes for connecting custom designs to the dual ARM cores, in addition to the main memory. We obtained this platform in 2015, and on top of it we built and tested the first and second prototype of Picos.

The second hardware platform is the SECO AXIOM Board [66]. When compared to the previous one, it has some exciting new features. First, it has a Zynq Ultrascale+ MPSoC Chip XCZU9EG-FFVC900 [86] which includes 4 ARM Cortex-A53 cores operating at a higher core frequency 1.1GHz than only 2 ARM Cortex-A9 operaing at 667MHz. Second, it has a much bigger FPGA and main memory. This allows the development of

hardware architectures that includes Picos and significant number of hardware accelerators to explore highly heterogeneous systems. In this board, we also tested out our designs with several choices of FPGA operating frequencies ranging from 50 to 300MHz. Finally, it has dedicated chips [45] for measuring the power consumption in real time during application executions. We obtained this platform in late 2016, on top of which we built and tested the third prototype of Picos.

3.1.3 HW Functional Accelerators

The HW functional accelerators (HwAccs) are designed in order to form highly heterogeneous systems, allowing tasks to be executed in different hardware devices. On one hand, they free us from the limited number of cores available in the existing boards and allow to construct highly diversified hardware structures to test Picos. On the other hand, they offer insight on how to integrate Picos with other types of hardware such as different processors, FPGA accelerators, and others as GPU. Finally, with the continuous advance in FPGA and FPGA+CPU technology, they can be also be seen as an alternative way to replace traditional processors for high performance and low power computing.

In this thesis, we annotated the C code tasks with High-Level Synthesis (HLS) directives to design different HwAccs. They are highly related to the applications, therefore a different HwAcc is generated for each different function in the applications.

Listing 3.1 shows an example of a functional accelerator that computes the tile matrix multiply (Matmul block) of a submatrix A per another submatrix B and adds the result in C. With this code, a HwAcc for Matmul block function with submatrices of size 32x32 (block size 32x32) can be generated. Afterwards, it can be integrated with Picos and the ARM cores.

There are three important sections in the code. First, the function interface definition from Line 1 to Line 8. It indicates that this function has three interfaces: FIFO interfaces `readyTask` and `finishTask`, and a AXI memory-mapped interface `data`. Second, the definition part for internal memory to hold matrices A, B, C from Line 10 to Line 13. At last, the third section is the actual funtional body part. In this section, this matmulBlock HwAcc first reads the addresses of the three matrices A, B and C, in the main memory. Then, three memcpy statements are used to read the submatrices using previous addresses. Afterward the matrix multiply computation is performed, and a memcpy is used to write the result submatrix C to main memory. Function matmulBlock is not shown here, but the details of the code are explained in Chapter 6, Section Background.

Listing 3.1: matmulBlock function with HLS directives for Picos++

```
1  void matmulBlockWrapper(hls::stream<uint32_t> &readyTask, float *data, \
2  uint32_t *finishTask){
3
4  //definition of interfaces
5  #pragma HLS interface ap_ctrl_none port=return
6  #pragma HLS INTERFACE m_axi port=data
7  #pragma HLS INTERFACE ap_fifo port=finishTask
8  #pragma HLS INTERFACE ap_fifo port=readyTask
9
10 //definition of internal memory for holding the matrices blocks A, B, C
11 float A[32][32]; float B[32][32]; float C[32][32];
12 static uint32_t taskInfo[6];
13 static uint32_t taskID_h, taskID_l, picosID, addrA, addrB, addrC;
14
15 //read addresses and ID information from Picos
16 StreamInData (taskInfo, readyTask, 6);
17 taskID_h = taskInfo[0];
18 taskID_l = taskInfo[1];
19 picosID  = taskInfo[2];
20 addrA    = taskInfo[3];
21 addrB    = taskInfo[4];
22 addrC    = taskInfo[5];
23
24 //copy data of matrices A, B, C from main memory to FPGA internal memories
25 memcpy(A, (const float *)(data + addrA/sizeof(float)), 1024*sizeof(float));
26 memcpy(B, (const float *)(data + addrB/sizeof(float)), 1024*sizeof(float));
27 memcpy(C, (const float *)(data + addrC/sizeof(float)), 1024*sizeof(float));
28
29 //compute
30 matmulBlock(A, B, C);
31
32 //copy data of matrices C from FPGA internal memory back to main memory
33 memcpy((const float *)(data + addrC/sizeof(float)), (const float *)C, 1024*sizeof(float));
34
35 finishTask[0] = taskID_h;
36 finishTask[1] = taskID_l;
37 finishTask[2] = picosID;
38 }
```

The HW functional accelerators used in Picos and the software-only runtime are slightly different. In the Picos runtime, the HwAccs receive a ready task from and send back a finished task to Picos, therefore simple FIFO interfaces are generated for those HwAccs described earlier. However, in the software-only runtime, HwAccs exchange ready and finished tasks with the ARM cores and therefore more complex AXI Stream interfaces have been used for connecting to ARM cores.

3.1.4 Operating System Support

The Linaro Embedded Linux OS with kernel version 3.19 has been used to operate on Zedboard, and the Ubuntu Embedded Linux OS with kernel version 4.19 has been used to operate on the AXIOM board. Picos and HW functional accelerators memories are mapped into the global address space, so Linux can recognize and interact with them. To successfully boot Linux in these hardware platforms, there are many steps to prepare after generating our custom designs, these steps can be found in Xilinx wiki pages.

3.1.5 Runtime Systems

Two runtime systems have been employed to manage the parallel execution of the applications when using the OmpSs programming model. One is the original software-only runtime Nanos++, and the other is the Picos accelerated runtime.

The Nanos++ [36] runtime is used as the baseline software-only runtime in this thesis. It natively supports the OpenMP task directives and the additional tasks constructs provided by OmpSs. As mentioned earlier in Chapter 2.1, an OmpSs task has typically six stages in the runtime. These stages are task creation, submission, ready, blocked, finalization and worker descriptor deletion. During task submission, the new task and all its dependences are submitted from the thread to the shared task dependency graph (TDG) in the runtime, and the runtime computes its relationships and inserts it in the right place. During task ready, the runtime schedules it to an idle thread for execution. During task finalization, when a task finishes execution, the runtime checks whether there are any successors of this task and updates the readiness of their dependences. These steps are slightly different in the Picos runtime.

Picos accelerated runtime is built on top of a modified implementation of Nanos++ [14], with the TDG management and task scheduling performed in hardware instead of software. In the Picos runtime, the task submission, ready and finalization are decoupled to be completed by the cooperation of both software and hardware. During task submission, the new task and all its dependences are sent directly to Picos either synchronously (described in Chapter 4) or asynchronously (described in Chapter 5). Then this task submission function returns and Picos tackles the computation and insertion of this task into the TDG in hardware. When a task is ready, Picos writes it into the main memory, then during task ready, the runtime simply checks if there are any tasks that are in the ready task pool in the main memory and dispatches them to the threads. Finally, during task

finalization, when a task executed in a thread finishes, it is sent directly to Picos either synchronously or asynchrously, and Picos updates and wakes up the successors. In order to perform this, a set of Picos APIs for Picos communication are embedded inside the Nanos++ libraries. By this way, the Picos runtime is transparent to application or even runtime programmers. The most important Picos APIs are shown below. The Picos APIs have evolved from the first to the third prototype, so there are some small technical differences between the different versions, but the following ones are representative and show the main common ideas. More details specified to each prototype can be found in Chapter 4, 5 and 6.

1 `picosInitialize(unsigned int buffersSize);` This function is used to initialize Picos with the size of the communication buffers that are going to be used to store the tasks. It includes allocating circular buffers for tasks and sending their addresses to Picos.

2 `picosShutdown();` This function is used to shutdown Picos, including HW reseting and deallocating all the communication buffers allocated in the initialization.

3 `picosSendNewTask(const *ptr task);` This function writes the task and dependences information of a new created task into the new task buffer.

4 `picosStatSendNewTask();` This function checks if a new task can be sent to Picos through the new task buffer.

5 `picosStatGetReadyTask();` This function checks if there is an available ready task in the ready task buffer.

6 `picosGetReadyTask(const *ptr task);` This function retrieves a ready task from the ready task buffer to the ready tasks pool for idle threads.

7 `picosFinishTask(unsigned int taskHandler);` This function puts the task information of a finished task into the finished task buffer.

8 `picosStatFinishTask();` This function checks if a finished task can be sent to Picos through the finished task buffer.

9 `picosSendExecTask(const *ptr task);` This function puts the task information of a ready task into the bypass task buffer.

10 `picosStatSendExecTask();` This function checks if a ready task can be sent to Picos through the bypass task buffer.

11 `setRegister(size_t id, uint32_t val);` This function sets the candidate register in Picos.

12 `getRegister(size_t id);` This function reads the value of the candidate register in Picos.

13 `picosPrintRegisters();` This function prints out all the debugging register values from Picos.

3.2 Benchmarks

This section shows the synthetic and real benchmarks[25, 79] that are specifically constructed and selected to show the capabilities of the different prototypes of the Picos system.

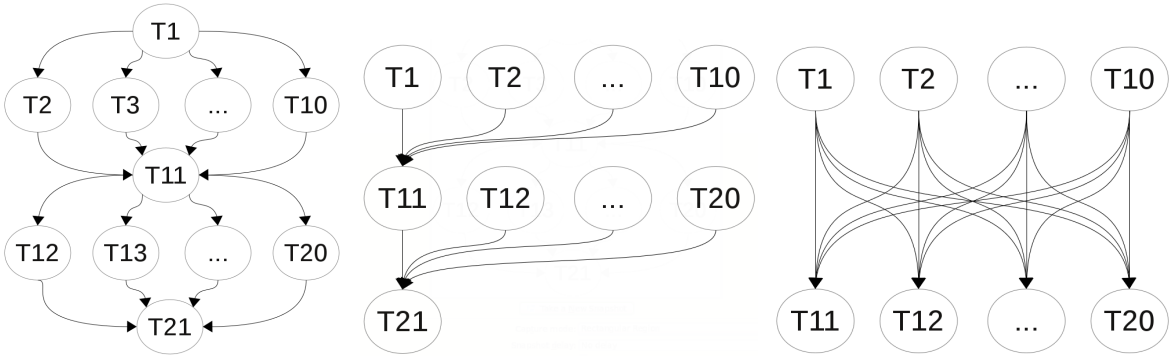
3.2.1 Synthetic Benchmarks

A brief description of the synthetic benchmarks follows. They have been carefully constructed to test the capabilities of the Picos system.

TestFree creates N tasks with M dependences. Each task has the same execution time T and all of them can be executed in parallel (all tasks are independent of each other). This benchmark is designed to illustrate the maximum processing capacity of the three prototypes of Picos, as it has the maximum parallelism among all the synthetic benchmarks.

TestChain creates N tasks with M dependences, each of a execution time T. Each task is the consumer of the previous task, and the producer of the next task. This benchmark is designed to show the worst case, as it has no parallelism at all and suffers the communication latency for offloading task dependences to hardware.

TestNested creates 16 parent tasks. The first 15 parent tasks have 2 dependences per task but they are independent from each other. Each parent task creates an inout chain of M child tasks, each child task can be configured to create it own nesting tasks. This inout chain is implemented by using the first dependence of each parent. The 16th parent task has 15 dependences that depend on all the previous 15 parent tasks by using their second



(a) Test-1P10C (b) Test-10P1C (c) Test-10P10C

Figure 3.2: Task dependence graphs of Test-1P10C, Test-10P1C and Test-10P10C

dependence. This benchmark with configurable nesting levels of tasks and configurable child task per parent is constructed to test the nested task support.

Test-1P10C creates N sets of 11 tasks. For each set, the first task is a producer task which has 10 dependences per task, the following 10 tasks are consumers that depend on each dependence of the producer task. Each task has the same execution time T. To facilitate the understanding of this benchmarks, we show the dependence relationships in Figure 3.2a.

Test-10P1C creates N sets of 11 tasks. For each set, the first 10 tasks are producers that have 1 dependence per task, the 11th task is a consumer with 10 dependences that depends on the previous 10 tasks. Each task has the same execution time T. The task-dependency graph is shown in Figure 3.2b.

Test-10P10C creates N sets of 20 tasks with 10 dependences per task. For each set, the first 10 tasks are producers with 10 output dependences each; the second 10 tasks are consumers where each task depends on all the first 10 tasks. Each task has the same execution time T. The task-dependency graph is shown in Figure 3.2c.

These last three synthetic benchmarks have been constructed to show the processing capability of the Picos designs for common complex dependences with different parallelism between TestFree (total parallelism) and TestChain (no parallelism).

3.2.2 Real Benchmarks

A representative set of real benchmarks in scientific computing and video/media decoding [25] [79] have been selected to show the abilities of task-dependence management, nested task support, and heterogeneous task scheduling in the Picos system during real executions. A brief description of their functionality follows:

Gauss-Seidel Heat is an iterative Gauss-Seidel solver for heat distribution.

LU Factorization decomposes an $m \times n$ matrix ($m >= n$) $A = LU$, with L unit lower triangular ($m \times n$) and U upper triangular ($n \times n$).

Sparse LU performs a LU decomposition over a square sparse matrix.

Cholesky Factorization is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose, as computes A = LL', with A an $n \times n$ matrix and L lower-triangular.

H264dec is a high performance H.264 video decoder, a video pedestrian_area.h264 is selected as input.

Multisort is a variant of the Mergesort, which sorts the input arrays using the divide and conquer method.

Matmul is matrix multiplication. It calculates the multiplication of two matrices $C = AB$.

Detailed configurations such as problem size and block size will be described in later chapters including evaluations of performance and energy consumption.

Both synthetic and real benchmarks have been compiled with the Mercurium source to source compiler, which translates the original C/C++ code of the benchmarks with pragmas for the task annotations. During execution, they invoke functional calls to either the Nanos++ runtime or the Picos++ runtime, used for the task-based programming model. The resulting code is compiled with cross compiler arm-linux-gnueabihf-gcc-4.5 or -5 for ARM 32bits (Zedboard) and arch64-linux-gnu-gcc-4.5 or -5 for ARM 64bits (AXIOM), with -O3 flags on when the applications are running on ARM cores only/SMP-only, with -O2[2]flags on when the applications are running on heterogeneous hardware.

3.3 Metrics

In this thesis, we aim at developing a hardware accelerated runtime Picos, which could improve the software-only runtime overheads especially on task dependence analysis, nested task support and heterogeneous task scheduling. In order to examine and evaluate its functionality and capability, four common metrics are selected to be used with different benchmarks. These four metrics are task and dependence repetition rates, performance, hardware resource consumption, and energy cost.

[2]O2 flag is used in this case because with O3 the compiler may emit some aarch64 instructions that are not valid when dealing with non-cacheable memory.

The task and dependence repetition rate are shown in cycles. They are measured first for the hardware manager only without any integration, and then repeated with the communication cost, and the runtime and OS overheads. This metric aims to show the exact processing abilities of Picos and also to give a general idea of the precise cost of Picos if it is intended to be integrated with different communication connection networks, with other runtimes or operating systems.

The performance is evaluated mainly with speedups, obtained by dividing the execution time of the sequential version by that of the parallel execution of the applications. Since the objective of this research is to show that the Picos hardware runtime can reduce the overheads caused by the default software-only runtime, all the speedups are obtained by using both the Picos runtime and a cutting-edge software-only one. Furthermore, for different prototypes different applications were selected in order to focus on examining the different capabilities of the hardware runtime. For example, for the first prototype, benchmarks with different and complex dependence patterns have been selected to show its task-dependence analysis ability. For the second prototype, special benchmarks have been selected to show the nested task support feature. Finally for the third prototype, besides the two purposes mentioned, additional benchmarks have been selected to show the heterogeneous task scheduling benefit.

In addition, to design such a hardware architecture, the hardware cost is without doubt a very important factor. Therefore all the prototypes shown in the thesis are presented together with their hardware cost. During the development, both the speed and area are important to consider when it comes to choose between different implementation methods.

Another goal of the Picos system is to be energy efficient. Energy is related closely to both the speed and area of the design. Therefore we measure the energy consumption of each application execution, and compare them with the software-only runtime and sequential executions.

These are the four main metrics used to evaluate the proposals of this thesis, detailed analyses are provided to explain the results showed in each chapter. Finally, note that the designs shown in this thesis are specific implementations based on FPGA. They can be implemented at a higher speed (operate at a higher frequency) by using more hardware resources, or the other way around, the presented designs show a good compromise between all these factors. Naturally, an ASIC implementation of the same designs would have a much higher operating frequency and a much smaller hardware cost as could be proved

by using exactly the same designs in different hardware design tools, but this aspect is out of the scope of this thesis.

Chapter 4

Task Dependence Manager -First prototype

Task-based programming models [59, 27, 3, 69, 39, 10, 52, 64, 12] are a very appealing approach to program multicore and many-core architectures as they are very simple to use yet very powerful. In these programming models the programmer exposes the available parallelism of an application by splitting the code in tasks and by specifying the data and control dependences between them. They are powerful as they are able to obtain high performance for a wide range of applications with a simple and clear set of annotations. With those annotations the runtime system manages the parallel execution of the workload, schedules tasks to available threads and synchronizes them. In addition, it decouples the application from the hardware platform, by applying optimizations such as locality-aware scheduling or data prefetching at the runtime system level in a generic and application-agnostic way [80, 22].

However, there are non-trivial disadvantages of their default software-only runtimes that prevent the exploitation of fine-grained parallelism. In theory, with fine-grained tasks within applications there is more parallelism to be exploited and this is easier to be balanced, thus leading to a higher performance. However, fine-grained parallelism usually does not lead to higher performance [71] and instead, task-based programming model behaviour is better exploiting coarse-grained parallelism in the applications. One of the main reasons is the software-only runtime system overhead, especially task dependence management, which increases significantly when exposed to a large amount of fine-grained tasks. With smaller sizes, there are much more tasks to be inserted into the task dependency graph. Indeed, there is much less time for the insertion as such tasks finish execution much faster. Additionally, threads can have a lot of contention when updating the aforementioned task dependency graph.

Figure 4.1 shows the speedup of OmpSs applications with a State-of-the-Art software-only runtime. Y-axis shows the speedup achieved when using task-based parallelism in

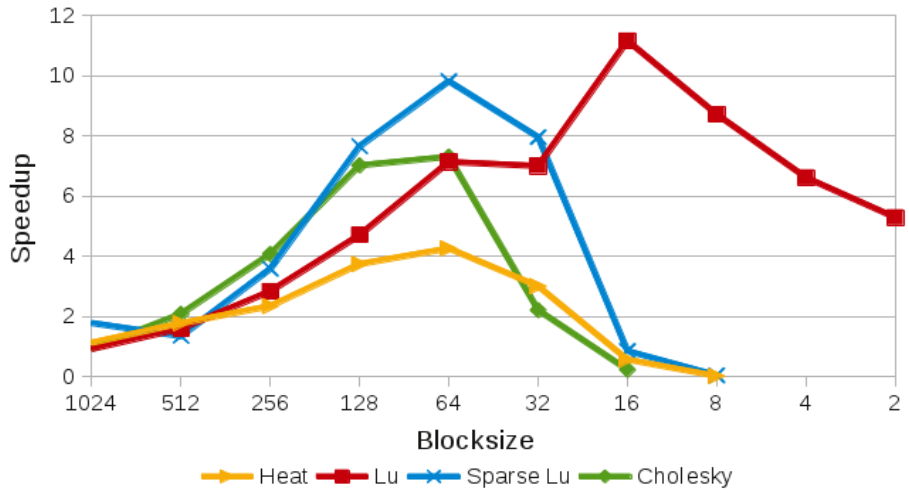


Figure 4.1: Speedup of OmpSs applications with software-only runtime with 12 threads. All the applications are with a fixed problem size 2Kx2K.

the applications: Heat, Lu, Sparse Lu and Cholesky, compared to their sequential code. X-axis shows the task granularity. In principle, the smaller the task size the more potential parallelism and easier load balancing. However, the speedup starts to rise at the beginning and then fall at the moment when the runtime overheads overtake the benefit of parallelism.

To overcome this deficiency and improve the performance, a straightforward and effective way is to reduce the software-only runtime overhead. Note that reducing the task creation overhead is important, nonetheless accelerating the task and dependence management is far more crucial [37, 14]. Task Superscalar [38] was proposed to accelerate task and dependence management using hardware. Its first VHDL prototype simulation analysis using ModelSim demonstrated high potential [88] by employing inter-task dependence analysis, dependence renaming and out-of-order execution. However, its straightforward hardware implementation presented unresolved deadlocks due to queue saturation and memory capacity. A new design called Picos [87] was proposed after a design space exploration using a simulator in C language to solve the Task Superscalar deadlocks and, in addition, to improve its performance and hardware resources. In this chapter, we present a hardware accelerator for task dependence management for task-based programming models for fine-grained parallelism. To the best of our knowledge, this first Picos prototype was the first sucessfully hardware task dependence manager implemented and integrated in a real system, and in particular, in an embedded system with an ARM-based SMP and a FPGA.

The main contributions of this chapter can be summarized as follows:

- A design exploration of different configurations of Picos with the objective of evaluating the best design to have the best task and dependence repetition rate.
- A proof-of-concept of functional hardware implementations for all the Picos configurations analyzed. All of them have been implemented on a Zynq 7000 SoC.
- Performance evaluation of all the hardware designs presented. This evaluation includes scalability for synthetic and real applications, resource consumption, and comparison to the current software runtime library of OmpSs.

4.1 Background

4.1.1 Task-based Programming Model

OpenMP provides a powerful way of annotating sequential programs with directives to exploit heterogeneity and task parallelism. For example in C/C++ language: `#pragma omp task depend(in: ...) depend(out: ...) depend(inout: ...)` is used to specify a task with the direction of its data dependences (scalars or arrays). Implicit synchronization between tasks is automatically managed by dependence analysis, and explicit synchronization is managed by using `#pragma omp taskwait`, which makes a thread wait until all its child tasks finish before it can resume the code execution.

We show an example of `multisort` source code with OpenMP annotations in Listing 4.1. Each multisort task instance can create four child multisort tasks as shown in lines 5-12 and three child merge tasks as shown in lines 14-19. The `taskwait` ensures that the calling multisort task (parent) has to wait for all its child tasks to finish before it can end.

When the compiler finds a task annotation in the program it outlines the next statement and introduces a runtime system call to create a task (represented as a Task Descriptor). At execution time, the runtime system manages task creation, computes task dependency graph, and schedules tasks when they can be executed because all their dependences are ready.

4.1.2 Software-only Runtime Overhead

OmpSs is a forerunner of OpenMP. It allows programmers to transform a sequential program into a parallel version with small effort, meanwhile ensures high performance with

Listing 4.1: multisort with OpenMP directives

```
1  void multisort ( size_t n, T data[n], T tmp[n]) {
2    int i = n/4L;  int j = n/2L;
3    if (n < CUTOFF) {sequential_sort(...);}
4    else {
5   #pragma omp task depend(inout: data[0], tmp[0])
6      multisort(i, &data[0], &tmp[0]);
7   #pragma omp task depend(inout: data[i], tmp[i])
8      multisort(i, &data[i], &tmp[i]);
9   #pragma omp task depend(inout: data[j], tmp[j])
10     multisort(i, &data[j], &tmp[j]);
11  #pragma omp task depend(inout: data[3L*i], tmp[3L*i])
12     multisort(i, &data[3L*i], &tmp[3L*i]);
13
14  #pragma omp task depend(in: data[0], data[i]) depend(out: tmp[0])
15     merge(i, &data[0], &data[i], &tmp[0]);
16  #pragma omp task depend(in: data[j], data[3L*i]) depend(out: tmp[j])
17     merge(i, &data[j], &data[3L*i], &tmp[j]);
18  #pragma omp task depend(in: tmp[0], tmp[j]) depend(out: data[0])
19     merge(j, &tmp[0], &tmp[j], &data[0]);
20  #pragma omp taskwait
21    }
22 }
```

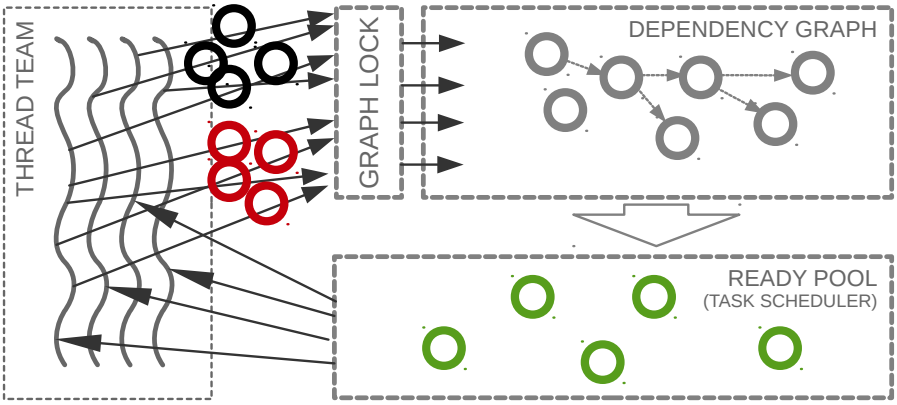


Figure 4.2: Shared task dependency graph access

applications by using coarse/medium granularity tasks. The OmpSs runtime (Nanos++) creates tasks, manages the inter-task dependences through constructing task-dependency graph, and enforces the correct task execution order. The task dependency graph is a shared memory structure that all threads in the system update concurrently whenever a task is created or once it is finished.

Figure 4.2 shows the runtime operational flow. The threads attempt to update the task dependency graph when they have a newly created task (red circle) or a finished execution one (black circle). When a new task arrives, its dependences are compared to those of all the earlier arrived tasks to determine its predecessors; when a finished task arrives, its dependences are used to update all its successors. Once all the dependences of a task are ready (green circle), it is copied to the ready pool and scheduled to the idle threads.

In the case of fine-grained parallelism, the number of tasks to be created increases and also, the task execution time is shorter. Those two aspects may impact the size of the task dependency graph, the contention inserting and updating it, and consequently, may lead to a performance degradation [71].

In addition, to ensure correctness the updates are protected with graph locks and only one thread can update the task graph at a given time. Once a thread gains a lock, several actions should be performed before it finishes its work and then it can release the lock so that another thread can obtain it. This can significantly reduce performance on many-core systems where a large amount of time is spent by workers waiting to get the lock and update the task dependency graph due to contention [14].

The next section describes the hardware architecture proposed to manage the task dependency graph. This architecture speeds up the task dependency graph management, reduces the execution time and the contention, and is more energy effective than the software-only runtime.

4.2 Picos Design Overview

The first prototype of Picos [71] aims to accelerate task dependency graph management for a large number of fine-grained tasks with complex patterns of dependences, for task-based data-flow programming models on multicore and many-core platforms. From the software aspect, it can be seen as a blackbox that (1) reads new tasks and their dependences from memory at task creation time; (2) writes ready-to-execute tasks back in memory for the worker threads and (3) reads finished tasks from memory to update the internal hardware task dependency graph.

Figure 4.3 shows a conceptual view of Picos organization. It is composed of five main components: one Gateway (GW), one Task Scheduler (TS), one Arbiter (ARB), and one Task Reservation Station (TRS) and Dependence Chain Tracker (DCT). Each pair of them are connected with one or more FIFO queues, and each has its own control logic, which only relies on the status (empty or full) and packets of those FIFO queues to ensure asynchronous communications with the others. This architecture is scalable by simply increasing the number of TRS and DCT instances. A design with four instances is able to manage up to 256 cores, and a baseline configuration with only one TRS and DCT is able to manage up to 8 cores without significant performance loss based on simulation results[87].

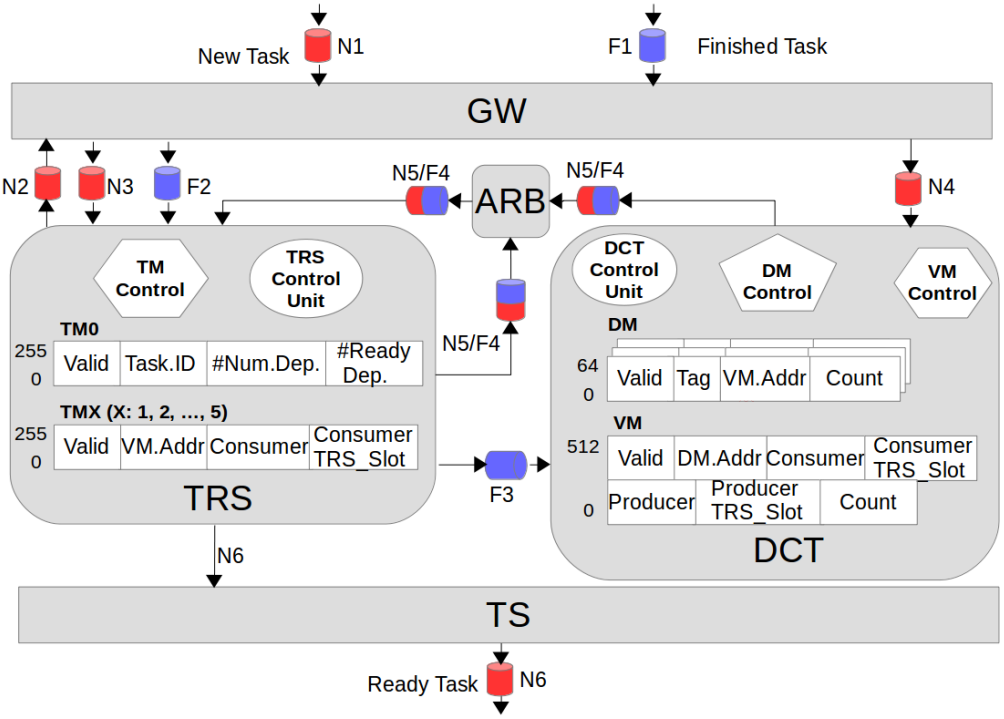


Figure 4.3: Picos task dependence manager

To begin with the explanation, it is important to know Picos exchanges three important types of tasks - new, ready and finished - with threads through three FIFO queues. The following text describes their specifications.

1 New task packet in the new task FIFO queue. It includes the task identifier (`TaskID`), the number of dependences (`#Num.Dep`), and the dependence memory address and dependence memory direction (input, output, and inout) for each dependence. In this implementation the number of dependences per task is limited to 15. However the software/hardware system is able to process tasks with any number of dependences as will be explained in the next chapter. The `TaskID` is used by the threads in the system to execute the task, the dependence memory addresses and directions are required for constructing the task-dependency graph.

2 Ready task packet in the ready task FIFO queue. It includes the `TaskID` and the Picos Task Identifier `PicosID`. `PicosID` contains `TRSID` (explained later) and the number of dependences of this task. As explained the `TaskID` is used by the threads to execute the task and the `PicosID` is sent back to Picos by the threads to notify the end of the task.

3 Finished task packet in the finished task FIFO queue. A finished task is represented

by the PicosID, which contains the TRSID and the number of dependences of the task. It is used to notify Picos, so it can update the internal task-dependency graph.

In the following, we will describe the five main components in details.

4.2.1 Gateway (GW)

GW is the first interface of Picos with the processing cores. It fetches new tasks and their dependences from the main memory, and requests a free TRS entry (represented by TRSID) from a TRS unit. Afterwards it dispatches the TaskID with the TRSID to the corresponding TRS, and the dependences with the TRSID to the DCT unit. It also fetches finished tasks from the main memory, and forwards them to the corresponding TRS.

4.2.2 Task Reservation Station (TRS)

TRS is the major task management unit. It stores all the in-flight tasks (up to 256), tracks the readiness of new tasks and manages the deletion of finished tasks. To manage the functionality of TRS, there are two main parts, one is the Task Memory shown in Figure 4.4, the other is the main control logic shown in Figure 4.5.

4.2.2.1 Task Memory (TM)

We first take a look at the TM design in Figure 4.4. It includes 6 independent internal memories (in the FPGA implemented as Block RAMs - BRAMs) named as TM Slot0 to 5, each slot has 256 entries. The TM control logic maintains a list of free memory entries. It accepts requests for a new free TM entry and recycles a released used one.

Each entry in TM Slot0 stores the TaskID, the number of dependences per task (#Deps), the number of ready notifications of dependences received from DCT (#ReadyDep) and a Valid bit. When the ready notifications equal to the number of dependences, the task in this entry is marked as a ready-to-execute task.

Each entry in TM Slot1 to 5 is used to store information associated with three dependences of the task in the corresponding entry in TM Slot0. Consequently the TM Slot1 stores information associated with the first three dependences and the Slot5 stores that of the last three dependences. The information of one dependence in each entry consists of the position of the dependence, the DCTID ("00", reserved for future use when we extend Picos to include 4 instances of DCT), the Version Memory

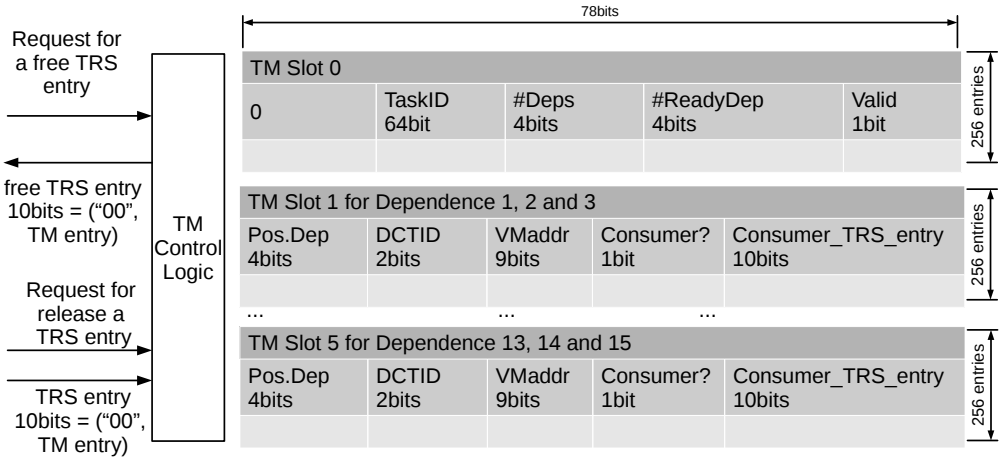


Figure 4.4: Task Memory organization

`address` (VM, explained later) where it is saved, the `Consumer` bit and the `Consumer task TRS entry`.

The TM control logic manages the 6 independent BRAMs as a set of 256 registers, each register has 6 78bits data that can be read in one cycle. Each TRS entry or TRSID is represented by 10bits where the lowest 8bits is used as TM entry to access one of the 256 entries, and the highest 2bits "00" are reserved for future use when we extend Picos to includes 4 instances of TRS module. For example, if a new task A enters and is assigned with TM entry 5, then its `TaskID` A and the number of its dependences are saved in TM Slot0 entry 5. After DCT analyzes all its dependences related to all those of tasks arrived earlier, DCT will notify the TRS. Then the aforemetioned information associated with its first three dependences are saved in TM Slot1 entry 5, that of its second three dependences are saved in TM Slot2 entry 5 and so on until that of its 13th to 15th dependences are saved in TM Slot5 entry 5.

4.2.2.2 Main Control Logic in TRS

As can be seen in Figure 4.3, TRS is connected to seven FIFO queues. For input, it mainly checks three FIFO queues from GW for new and finished task and from the Arbiter unit (ARB) for communication dependence packets. For output, it outputs free TRS entries for new tasks to the GW, communication dependence packets to the ARB, finished dependence packets to DCT, and ready tasks to the Task Scheduler (TS). Communication dependence packets are used between TRS and DCT and they are managed by the ARB module (explained later in Section 4.2.3).

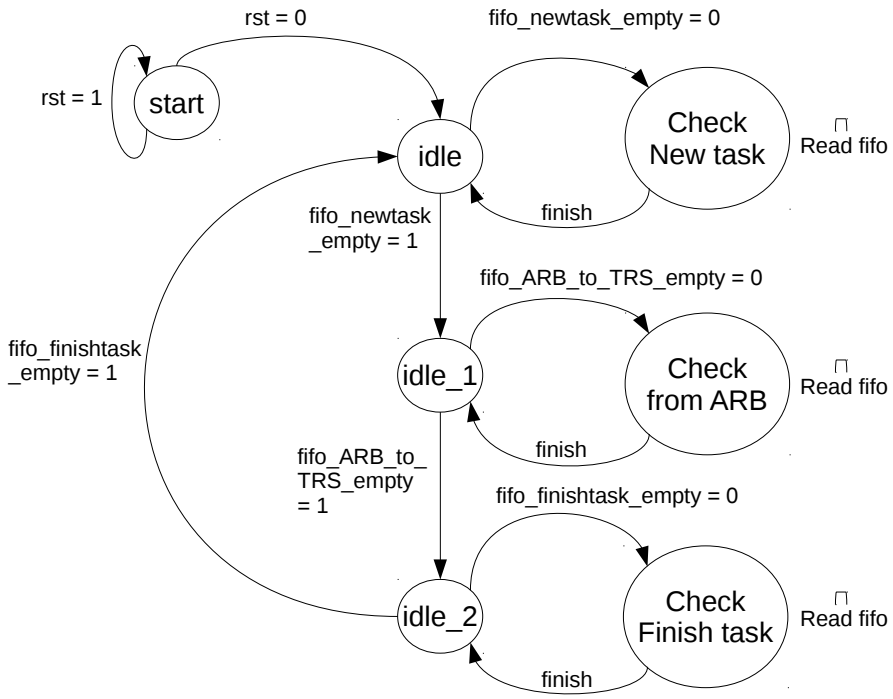


Figure 4.5: TRS control logic in Figure 4.3

Figure 4.5 shows the main Finite State Machine (FSM) for the TRS module. As can be seen, the reset signal sets the state machine to `IDLE`. Then if the new task FIFO queue is not empty, TRS enters state `Check New task` and sets a read FIFO queue signal to read a new task packet (this packet including a new task and all its dependences). When a new task and all its dependences are read from the FIFO queue, they are processed. When the processing finishes, a finish signal is set and the state machine jumps back to `IDLE`. On the other hand, if the new task FIFO queue is empty, and the FIFO queue from ARB to TRS is not empty, TRS sets a read FIFO queue signal to read a communication dependence packet and enters state `Check from ARB`. When a communication dependence packet is read and processed, a finish signal is set and the state machine jumps back to `IDLE_1`. Else if the finished task FIFO queue is not empty, TRS enters state `Check Finished task` and reads a finished task packet. Afterwards it goes back to check for new task.

Listing 4.2 describes the actions completed during the state `Check New task`. As we have mentioned earlier, new task packet includes `TaskID`, `#Deps`, ..., etc. In our system, since each main module is decoupled from the others through FIFO queues, there is a possibility that the new task packet from GW might come later than the communication dependence packet from ARB to TRS, which changes the TM Slot0 entry. Therefore

Listing 4.2: State Check New task in Figure 4.5

```
1  Check New task:
2  - IDLE:       if (read_ack = 1) then
3                   read the corresponding TM Slot0 entry;
4                   check din_fifo_newtask;
5                end;
6  - Check_DIN:  save TaskID, #Deps and set the Valid bit to 1 in the TM Slot0 entry;
7                if ((#Deps = 0) or (#Deps != 0 and #Deps = #ReadyDep)) then
8                   mark this task ready and send to TS;
9                end
10               jump back to IDLE;
```

Listing 4.3: State Check from ARB to TRS in Figure 4.5

```
1  Check from ARB:
2  - IDLE:       if (read_ack = 1) then
3                   check din_fifo_ARB;
4                end;
5  - Check_DIN: if (it is an empty ready packet) then
6                   #ReadyDep++ in TM Slot0 entry;
7                   save VMaddr in the corresponding TM Slot1-5 entry;
8                elsif (it is a chained ready packet) then
9                   #ReadyDep++ in TM Slot0 entry;
10                  if(the TM entry has a chained task) then
11                     send a chained and ready packet to ARB;
12                  end
13               elsif (it is an empty update) then
14                  save VMaddr in the corresponding TM Slot1-5 entry;
15               elsif (it is a chained update) then
16                  update the Consumer and Consumer_TRS_slot in TM Slot1-5 entry;
17               end;
18
19               if(#Deps = #ReadyDep) then
20                 mark this task ready and send to TS;
21               end;
22
23               jump back to IDLE;
```

we always read that entry first before modifying it as can be seen in states `IDLE` and `Check_DIN` in listing 4.2.

Listing 4.3 decribes the actions completed during the `Check from ARB` state in the main FSM. There are four different types of communication dependence packets necessary to manage the communication between TRS to TRS, and TRS with DCT. The details of these communication dependence packets are described in Section 4.2.2. As can be seen, each packet here can be an empty ready, a chained ready, an empty update or a chained update, each different packet corresponds to different actions. In this state, we also check if the number of dependences is equal to that of the ready dependence notifications received from DCT. If they are, we mark the task ready and send to TS. Listing 4.4 describes the actions completed during the `Check Finished task` state in the main FSM. As we mentioned earlier, Finished task packet includes PicosID, which consisting of TRSID and #NDeps. For each dependence, TRS sends a finished dependence packet

Listing 4.4: State Check Finished task in Figure 4.5

```
1  Check Finished task:
2  - IDLE:       if (read_ack = 1) then
3                   check din_finishtask;
4                end;
5  - Check_DIN: read TM Slot0 for #Deps:
6                for 1:#Deps loop
7                    read corresponding TM Slot1-5 entry and obtain VM address;
8                    send a finished dependence packet to DCT;
9                end
10               jump back to IDLE;
```

bit40 bit0

Chained 1bit	Ready 1bit	TRSID 10bits	Pos. Dep 4bits	DCTID 2bits	VM addr 9bits	Chained TRSID 10bits	Chained Pos. Dep 4bits

Figure 4.6: FIFO packet from TRS to TRS, DCT to TRS through ARB

to DCT, this packet only contains VM address.

In Picos, the tasks are organized in the producer-consumer (out/inout-in) and the producer-producer (out/inout-out/inout) chains. The producers and the last consumer are always saved in DCT, while all the previous ones are saved in TM using the consumer sections as can be seen in Figure 4.4. In the producer-producer chain, the tasks are awaken in sequence, while in the producer-consumer chain, the tasks are awaken in a reversed way from the last consumer. The mechanism for waking up dependences dynamically can be tricky to understand, therefore there is a separate Section 4.3.2 dedicated for this explanation.

4.2.3 Arbiter (ARB)

The ARB module reads the packets from TRS and from DCT in turns, and writes them into the FIFO queue to TRS. There are four types of communication dependence packets necessary to coordinate the TRS and DCT. Figure 4.6 shows the specification of the communication dependence packets. It is composed of Chained, Ready, TRSID, Position of Dependence, DCTID ("00", reserved when we extend Picos to have 4 instances of DCT), VM address, Chained TRSID and Chained Position of dependence. The last two fields are only used by the chained update dependence packet as explained in the following list. The `(Chained, Ready)` bits are used to represent four different types of communication dependence packet.

1 `Type "01"` indicates an empty ready dependence packet. It usually happens to be the first time a dependence has ever appeared in DCT. In this packet, only limited

areas are used. The TRSID is pointing to the task that is associated with this dependence in TM Slot0. The Pos.Dep indicates it is the N-th dependence of the task and the location of the dependence in TM Slot1-5. The VM address in the figure indicates where the information of this dependence is saved in VM.

2 `Type "11"` indicates a chained ready dependence packet. The chained ready is to address when this dependence is not the first consumer, and when the producer in this producer-consumer chain has finished. This packet is sent from TRS to itself (or to other TRS units if they exist in the system) through the ARB module.

3 `Type "00"` indicates an empty update dependence packet. This usually happens for the first input consumer or a producer dependence.

4 `Type "10"` indicates a chained update dependence packet. This happens when the dependence is in a consumer chain by DCT, while the producer (output) dependence is not ready.

4.2.4 Dependence Chain Tracker (DCT)

DCT is the major dependence management unit. It manages task dependences through one Dependence Memory and one Version Memory (DM and VM respectively). Their specifications in Figure 4.7. For each new arrived dependence, DM performs hashes and compares (address match) the dependence address to look for it among the tags of all these arrived earlier. DM saves it in the matched place and notifies their relations to VM and to TRS. VM receives those notifications from DM and keeps all the references (consumer and producer matches) to the same dependence address (called versions). Otherwise, the dependence with unique memory address is allocated with a new space and is saved as a tag. TRS receives those notifications - communication dependence packets - and processes them (can be seen in Section 4.2.2 Listing 4.3).

4.2.4.1 DM Designs

For each new dependence entering DCT, DM performs address match for it to these arrived earlier, to establish data dependences. Later, for each finished dependence from TRS, DM read/write are performed on DM for releasing data dependences. Both processes such as finding a place to save the dependence, and matching addresses are critical for the prototype performance. On one hand, each task can have multiple dependences

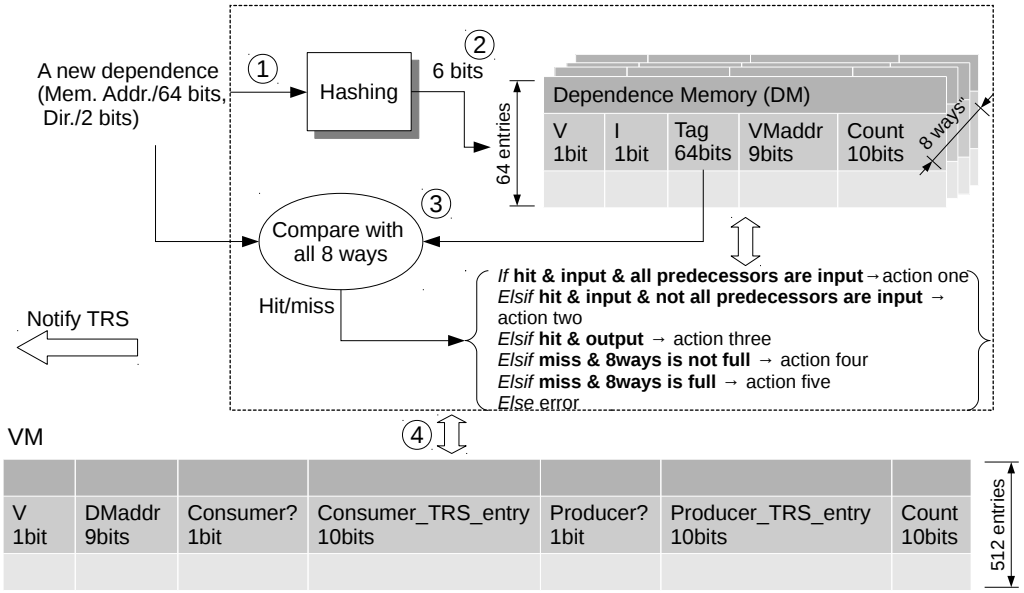


Figure 4.7: DCT Memories organization

that stresses DCT more than TRS; on the other hand, the whole system may stall if new dependences cannot be stored in DM which can be due to DM conflicts (more than 8 dependences try to use the same DM entry in the case of 8-way) or memory capacity. To overcome possible conflict stalls meanwhile lowering hardware cost, three different designs of DM are proposed and evaluated as one of the many trade-off implementation decisions in Picos:

- DM 8way: a 64-entry, 8-way associative cache-like memory with direct hash.
- DM 16way: a 64-entry, 16-way associative cache-like memory with direct hash.
- DM P+8way: a 64-entry, 8-way associative cache-like memory with Pearson hashing[61].

Figure 4.7 shows a simplified diagram of one of the DM designs: DM P+8way. The other two designs are identical if the Hashing unit is removed. The memory is implemented by using 8 independent bank memories (BRAMs in case of using a FPGA implementation) with 64 entries each, therefore, with 6-bits entry address, we can access it and obtain the context of all 8 ways in 2 cycles; in order to access a specific way, 3 additional address bits are required. Each DM way comprises the Valid bit (V), the Input bit (I), the Tag and the Data (VM address, count). The Valid bit indicates the way is occupied or free. The Input bit (I in the DM entry in Figure 4.7) indicates that all the dependences arrived earlier in this way are of input direction (dependences with the same memory address are

saved in the same DM way), thus they are independent as they all indicate consuming data from the same memory address. The count value in a DM way indicates the number of appearances of the same dependence address while the count value in a VM entry indicates the number of consumer dependences that are dependent on this entry. The VM address saved in each DM way points to the latest VM entry of the dependence with the same memory address, also known as the latest version.

DM supports three operations: read, write and compare. DM read/write are general RAM memory read/write operations. For DM read operation, on the first cycle the user provides address and asserts read enable signal, one the second cycle, the user gets the output result. For DM write operation, the user provides address and data, asserts write enable signal and on the next cycle the data is written into the designated location. For easier understanding, the DM comparison operation is similar to cache look up. With DM compare operation, the Pearson hashing function is first applied to each 8bits of the 64bits to randomize the value of the dependence address; and then the least significant 6bits after the xor of these hashing values are used to index the 64 entries of memory; one cycle later the whole 64bits dependence address is used to compare with the tags of all 8 way outputs of the entry. Depending on the dependence direction (Dir), Hit/Miss results from comparing with all 8 ways, we either obtain the Hit DM way and VM entry addresses for updating actions or a new DM way and VM entry for saving the new dependence. When there is a miss and there are multiple free ways inside an entry, way 0 has the highest priority and way 7 has the lowest priority for being selected as a free DM way.

Memory addresses of dependences always tend to group in clusters for certain applications, and the addressing of DM 8/16way configurations leads to large amount of conflicts that stall the design. By applying Pearson hashing, the memory conflicts are expect to reduce and thus greatly speedup the dependence management.

4.2.4.2 Version Memory

Figure 4.7 also shows a simplified diagram of the VM design. VM is a 512 entry memory, implemented using a BRAM in the FPGA with 512 entries of 42 bits each. VM is managed as a set of 512 registers. Similarly to the TM control logic, the VM control logic also maintains a list of free and occupied entries. It accepts free entry request and recycles used ones.

4.2.4.3 Operational Flow in DCT

DCT mainly processes new and finished dependence packets. In Figure 4.7 **when a new dependence packet enters** (step with circle 1 in the figure), first all the 64bits of its memory address go through the hashing module to obtain a 6 bits entry address (step with circle 2 in the figure). Then the 64 bits are used to compare with all the `tags` from the 8 ways (step with circle 3 in the figure). Depending on the DM comparison results, different actions named as action one to five are performed (step with circle 4 in the figure).

If it is an input and got a hit in one of the 8ways, additionally the `Input bit I` was set to 1, action one is performed. In DM and VM, the `count` value is increased by 1. Then an empty ready dependence packet is sent to TRS through the ARB module. However if the `I` bit was not set, action two is performed. First in DM and VM, the `count` value is increased by 1. Second, if the `consumer` bit in the VM entry had been set to 1, it means the current dependence is not the first consumer of a previous producer dependence. In this case, a chained update packet is sent to TRS with the `TRSID, Pos.Dep` updated with the new dependence, and the `Chained TRSID, Pos.Dep` with the previous one in VM. Otherwise, it is the first consumer, an empty update packet is sent to TRS. Finally, in VM, the `Consumer_TRS_entry` is updated with the new one associated with the new dependence.

If it is an output and got a hit, action three is performed. First, in the hit DM way, the `count` is increased by 1. In VM, a new entry is allocated with the `count` value initialized with 1 and the `DM address` value updated. Second, in the old VM entry, the `count` value remains the same, and we set the `Producer` bit to 1 and update the `Producer_TRS_entry` with the TRSID associated with this new dependence. If the `I` bit was set to 1 in the hit DM way, we also set it to 0 now. Finally, an empty update packet is sent to TRS.

If it is the first time that the dependence appeared in DM, and the 8 ways are not full, action four is performed. A new DM way and a new VM entry are allocated. If there are multiple free ways in the DM entry, a new DM way with the smallest address is allocated. An empty ready packet is sent to TRS. if the dependence is input, the I bit will be set to 1, otherwise it remains at 0.

If there is a miss and the 8 ways are all occupied we consider there is a DM full and action five is performed. DCT stops reading and processing new dependences, and continues reading and processing finished dependences until DM has an empty slot to process the one that caused this full situation.

When a finished dependence packet enters, the associated VM address is first used to read the VM entry, and then the DM way. In the VM entry, if the `Consumer` is set, DCT sends a empty ready packet to TRS to wake up the last consumer task in chain, and the `counts` in both VM and DM are decreased by 1. When TRS receives this communication dependence packet, it checks if the task is ready and also if any of its dependences had previous consumer tasks to be waken. If so, TRS sends an empty ready packet to the ARB module. When all the consumer tasks have eventually finished, the count value in VM will be reduced to 0; when it is 0, DCT checks if the VM entry also has the `Producer` bit set. If it was set to 1, DCT sends an empty ready packet to TRS to wake up the next producer task. When the `count` reaches 0 in the DM way or VM entry, they are freed and recycled.

4.2.5 Task Scheduler (TS)

TS is the second interface between Picos and the processing cores. It stores all ready tasks and schedules them to idle workers.

4.3 Operational Flow of Picos

4.3.1 New and Finished Task Processing

Picos consists of two major procedures: new and finished task processing as shown in Figure 4.3.

When a new task arrives (N*): GW reads its meta-data and dependences (N1). Then it checks for a free TRS entry. If there is not any, the GW does not process the new task, instead it starts to process finished tasks until a TRS entry is freed (which happens after the first finished task is processed). Otherwise, it obtains one free TRS entry (N2) and dispatches the new task to TRS (N3). If the new task has dependences, GW forwards each of them with the TRS entry to DCT (N4). The TRS entry will be used by DCT to index the task that owns the dependence.

Once TRS receives this new task from GW, it saves its TaskID and number of dependences inside the assigned TM Slot0 entry. If it has no dependences (`#Num.Dep.` = `#Ready Dep.` = `Zero`), TRS marks it ready and sends it to TS for execution (N6); otherwise, TRS will wait for notifications (ready or dependent, N5) from DCT for each dependence. For each ready notification, TRS increases the corresponding `#Ready Dep.`

by 1 in TM Slot0; and for each communication dependence packet, TRS saves it in the corresponding TM SlotX (X: 1, 2, 3, 4, 5) entry. TRS only marks the new task ready after all its dependences are ready.

When DCT receives the dependences. For each of them, DCT checks whether it is dependent on those arrived earlier (N5), and saves it in both DM and VM. If the dependence is not dependent on previous ones, DCT sends a ready packet (the corresponding TRS entry and VM address) to TRS; otherwise DCT sends an update packet (TRS entry, VM address, dependent TRS slot) to TRS. The VM address is to be used by TRS later to reference the location of this dependence in DCT.

When a finished task arrives (F*): GW reads the finished task (F1), and then distributes it to TRS (F2). Once the TRS receives the finished task, firstly it checks TM Slot0 entry for `#Num.Dep.` then examines this `#Num.Dep.` of dependences in TM SlotX; secondly it sends finished packets (VM address) for each dependence to DCT (F3); finally it deletes the task inside the assigned TM entry and sends a release TM entry request to the TM control logic.

When DCT receives finished dependences packets. For each dependence, DCT checks the corresponding VM entry (Consumer, Producer, Count) to see if there are other ones that depend on it. If there exist no such dependences, DCT deletes it from the DM and VM directly; otherwise DCT keeps on tracking of its consumers/producers, and send ready dependence packet to TRS to wake up the last consumer or the next producer task in the chain (F4). When TRS receives this message, it first checks in TM Slot0 (`#Num.Dep, #Ready Dep`) whether this task is ready or not. Then it checks the corresponding TM SlotX consumer section to see if there is another task that depends on this dependence and sends a message to wake it up (F4). Once all such dependences that depend on it are resolved and finished, the dependence is deleted inside the VM and DM eventually. The detailed communication flow have also been explained in Section DCT, moreover it is also shown in the following example.

4.3.2 An Example of Dependence Chains

To dynamically establish dependences and wake up tasks rapidly and economically is challenging and crucial for system performance. Figure 4.8 shows an example of six tasks, where each task has only one dependence (A) with different directions. This example assumes that all the described tasks arrive to Picos before the first task finishes its execution. The six tasks form a mixed Producer-Consumer chain (Task1, 2, 3 and 4)

and Producer-Producer chain (Task1, 5, 6) established inside TRS and DCT. Each solid line shows how the dependence chain is established according to the sequence of new tasks. The dashed line (with labeled number) shows the order in which they are woken up after Task1 finishes. Note that the Producer-Consumer chain is woken up from the last consumer, the Producer-Producer chain is woken up in sequence.

```
// Pseudo code
Task1(output(A));
Task2(input(A));
Task3(input(A));
Task4(input(A));
Task5(output(A));
Task6(output(A));
```

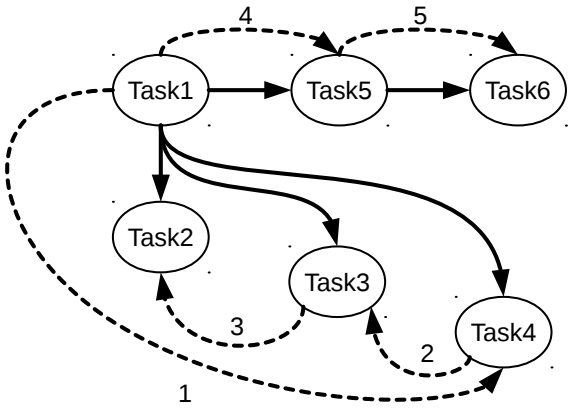


Figure 4.8: An example of dependence chain

When Task1 arrives, its dependence is forwarded to DCT. For the first task as its dependence is independent, a new DM way and VM entry are assigned to it. The DM way stores the memory address of the dependence as Tag, and the VM entry stores consumer/producer related information. In each DM way, it saves this VM entry address; similarly this DM address is saved inside the VM entry. In addition, in each DM way, there is a 10-bits counter to count the total appearances of the dependence with the same memory address. In each VM entry, there is a 10-bits counter to count the total appearances of the consumer dependences with the same memory address. Therefore with this first dependence A of Task1 enters, the counters are assigned with 1. For this dependence, DCT sends a ready message to TRS. TRS then saves the VM address of this dependence inside the assigned TM SlotX entry, marks it ready and sends it to TS for execution.

When Task2 arrives, its dependence is forwarded to DCT. Once DCT receives the dependence, it first does a DM compare and realizes that it is the first consumer. In this case, it is saved in the same DM way and VM entry (increase the count of this dependence to 2, and update the Consumer TRS entry of Task1 with Task2 inside VM). An empty update message is sent to TRS.

When Task3 arrives, DCT detects that its dependence is the second consumer. The dependence is then saved in the same DM way and VM entry (increase the count of this dependence to 3, and update the Consumer TRS entry of Task2 with Task3 inside VM).

At the same time DCT notifies TRS that Task2 will be waken up after Task3 through a chained update message. The same happens for Task4 (increase the count of this dependence to 4, and update the Consumer TRS slot section in VM). In this way the last consumer is stored in DCT while the former ones are kept chained in TM SlotX entry of the previous task inside the TM. Until now one DM way and one VM entry have been assigned, and the count in both DM way and VM entry are 4.

When Task5 arrives, DCT detects that it is the fifth time when the A dependence appears and it is a producer. A new VM entry is assigned to store this latest version of producer, the count is initialized with 1. In the old VM entry, this dependence A is saved in the Producer section. Both VM entries point to the same DM way, while in the DM slot it updates to point to the new VM entry. An empty update message with the new VM address is sent to TRS. Now there are one DM way and two VM entry allocated in total. In the DM way, the count is 5, in the old VM entry the count is 4, and in the new VM entry the count is 1. The same happens for Task6 to keep the producer-producer chain in DCT. Up to this point, one DM way and three VM entries have been assigned. The count in the DM way and the three VM entries are 6, 4, 1 and 1 respectively.

When Task1 finishes, TRS notifies DCT of the finish of the first A dependence. DCT checks the corresponding VM entry and sends a ready message to TRS for Task4 (dashed line with label 1 in Figure 4.8). Meanwhile the count in the DM way and the first allocated VM entry are decreased by 1. Once TRS receives this message, it wakes Task4 and sends another ready message (managed by the Arbiter module) to wake Task3 (dashed line 2), and then Task2 (dashed line 3). Now Task2, 3 and 4 are marked ready after TRS receives these three ready messages and are sent to TS then be scheduled to execute in idle workers. Whenever a task finishes, TRS notifies DCT. DCT decreases the count value in the DM way and the first allocated VM entry. Once DCT receives three more finished messages from Task4, Task3 and Task2, it wakes up Task5 (dashed line 4). Meanwhile the count DM way and the first allocated VM entry are 2 and 0, therefore it deletes the first VM entry of the dependence.

When Task5 finishes, DCT wakes up Task6, meanwhile the count DM way and the second VM entry are decreased by 1. So now the value in both places are 1 and 0, therefore the second VM entry is deleted. Finally after Task6 finishes TRS notifies DCT to delete both the DM way and the third VM entry.

4.4 Experimental Setup and Methodology

This section describes the experimental setup, the testing platform for the first prototype of Picos, and the synthethic and real benchmarks used for evaluating the hardware task dependence manager.

4.4.1 Experimental Setup

Xilinx ISE Design Suite 14.4, Vivado 14.4, SDK and a Zynq 7000 SoC Platform (Zedboard) are used to develop the Picos prototype and its embedded system. Zedboard includes one FPGA Chip XC7Z020-CLG484[85] which comprises the Dual ARM Cortex-A9 MPCores and a FPGA part, also know as the processing system (PS) and the programmable logic (PL) part. The OmpSs programming model in use in this work to program the benchmarks is supported by the Mercurium compiler 1.99 [42] and the associated Nanos++ RTS [17].

Sequential and parallel execution time, in addition to execution traces, of OmpSs applications from the software-only implementation are obtained from a shared memory machine which has up to 12 cores (Zedboard was not used for obtaining workloads because it has only two cores). The shared memory machine has 2 NUMA nodes with 1 socket each, each node has 64GB main memory. Each socket is a Xeon E5-2630L with 6 cores with dynamic frequency control up to 2.0GHZ. Each core has 2 threads sharing resource. In total, we can use 12 cores and up to 24 threads with hyper-threading.

Sequential and parallel execution time of the same applications from the Picos prototype are obtained in Zedboard by using the information of aforementioned traces. Traces include task creation and execution time in cycles, task identification, dependence addresses and directions. The task creation and execution time in cycles is obtained through instrumenting the sequential execution in the shared 12-core memory machine described above. In Zedboard, there are one counter for task creation time, and another 24 counters served as 24 workers for task execution time. When the value of the task creation counter is equal to the task creation time of a task, this new created task and its dependences are sent to Picos. When a ready task is scheduled to a worker by Picos, this worker counts until the task's execution time and then notifies its finished status to Picos. In this way, although task creation and execution are simulated in the ARM cores in Zedboard, Picos, is executed for real as a co-processor synthesized in the FPGA in Zedboard. Traces are also used to feed a Perfect simulator which measures critical-path task execution to show

the roofline speedup of each OmpSs application.

All the speedup shown in this paper is computed against the sequential execution time.

4.4.2 Hardware Testing Platform

The embedded system also named Hardware-In-the-Loop (HIL) Simulation Platform is a modern way to validate the functionality and examine the performance of IP-cores. Figure 4.9 shows the organization of the HIL Platform developed for testing the first prototype of Picos hardware task dependence manager. The PL part uses a 80 MHz global clock, and a 64bits AXI Timer synchronized with the same clock as the global timer. In the PS part, the two threads are operating at 667 MHz. In this section, we present two major operational modes of the platform:

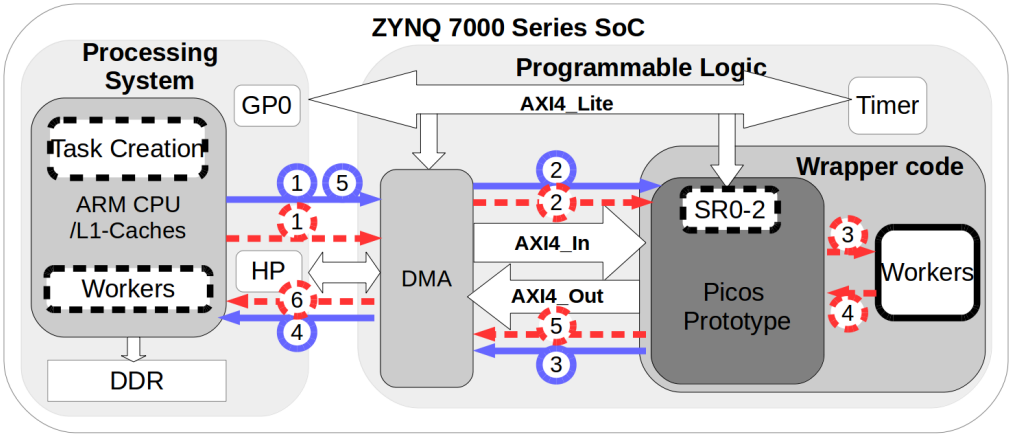


Figure 4.9: Hardware-In-the-Loop Platform

HW-only (Dashed labeled line): employs a naive process as all the tasks are sent to Picos once (1, 2), and all the finished tasks are retrieved all at one time (5, 6). Workers are implemented inside the PL part so that ready tasks can start executing shortly after there are idle workers (3) and finished tasks are sent back to Picos for updating its internal task-dependence graph (4). This mode is chosen to show the task and dependence repetition rate without any scheduling or communication overheads.

Full-system (Solid labeled line): employs a close-loop process. Each task is created and sent to Picos for dependence analysis (1, 2); ready task is retrived from Picos to the ARM core for execution in the workers (3, 4); finally finished task is sent back to notify Picos (5) to carry on the process until the last task. Each message between Picos and ARM core carries one task at once.

Three queues are employed inside the Picos prototype for new, ready and finished tasks; and SR0-2 are the corresponding status registers. The communication latency for sending or retrieving data via AXI Stream interface takes around 200 to 300 cycles for each message. When comparing with the **HW-only** mode, the **Full-system** mode is much more closer and accurate to the Picos system we proposed, as it takes into consideration the data exchange scheme between ARM cores and FPGA, moreover the task creation and executions are simulated in the ARM cores.

4.4.3 Benchmarks

Both synthetic and real applications are chosen to evaluate the Picos prototype.

Synthetic benchmarks including TestFree, TestChain, Test-1P10C, Test-10P1C and Test-10P10C are selected. The first one is designed to illustrate the maximum processing capacity of Picos, since all its tasks can be executed in parallel (tasks are independent of each other). The second one is designed to show the worst case in Picos, since it has no parallelism at all, additionally it has all the communication overheads for offloading all the task dependence analysis to Picos. The following three are designed to show the ability of Picos by using some common complex dependence patterns that are not as free as TestFree while are not as rigid as TestChain. Their functionality description can be found in Chapter 3, Section 3.2.

Real applications Gauss-Seidel Heat, Lu, Sparse Lu, Cholesky[25] and H264dec[79] are selected to study the performance and detect possible bottlenecks in Picos prototype. Their functionality description can be found in Chapter 3, Section 3.2.

Table 4.1 shows basic information about these real benchmarks obtained on a shared memory machine. For each benchmark, the table shows, from left to right, its problem and block size (for H264dec, 10f stands for 10 HD frames), number of tasks, number of dependences per task, average task size and the sequential execution time in cycles respectively.

4.5 Results

This section evaluates the first prototype of Picos design. First, both the performance and hardware cost of three different DM designs are analyzed to select one that balances both. Next the task and dependence repetition rates of Picos are evaluated by using synthetic

Table 4.1: Characteristics of tasks in real benchmarks

Name	Problem/ Block size	#Tasks	#Dep	Ave.Task Size in cycles	Seq. Exec. Time in cycles
Heat	2048/256	64	5	3.51e+06	2.25e+08
	2048/128	256		8.20e+05	2.07e+08
	2048/64	1024		2.17e+05	2.11e+08
	2048/32	4096		7.19e+04	2.41e+08
Lu	2048/256	36	2	5.67e+07	2.04e+09
	2048/128	136		1.49e+07	2.04e+09
	2048/64	528		4.13e+06	2.17e+09
	2048/32	2080		1.53e+06	3.18e+09
SparseLu	2048/256	34	1-3	2.74e+07	9.30e+08
	2048/128	212		4.36e+06	9.24e+08
	2048/64	1512		6.47e+05	9.78e+08
	2048/32	11472		8.28e+04	9.50e+08
Cholesky	2048/256	120	1-3	6.63e+06	7.61e+08
	2048/128	816		9.71e+05	7.89e+08
	2048/64	5984		1.47e+05	8.77e+08
	2048/32	45760		2.94e+04	1.34e+09
H264dec	10f/8	2659	2-6	2.06e+06	5.48e+09
	10f/4	9306		5.91e+05	5.50e+09
	10f/2	35894		1.53e+05	5.48e+09
	10f/1	139934		3.94e+04	5.51e+09

benchmarks. Finally, a much thorough scalability study of Picos with the most balanced DM design is shown by using real benchmarks.

4.5.1 Difference between DM Designs

To decide the best implementation, we first check the performance of all three DM designs with up to 12 workers. Figure 4.10 shows their speedup (bar, y-axis) by using four real benchmarks under the HW-only model. Each benchmark is shown with two representative block sizes. The speedup is calculated by comparing the sequential execution time with the parallel execution time of the same benchmarks with the same problem and corresponding block sizes.

For Heat and Cholesky, Picos P+8way achieves the best speedup and scales well from 2 to 12 workers. For example with Picos P+8way, Heat with blocksize 64 scales from 2x to 5.9x, Cholesky with blocksize 128 scales from 2x to 11.5x. While with the other two designs, Heat achieves up to 1.2x and Cholesky achieves up to 4x speedup in total.

For LU and SparseLU, Picos P+8way and Picos 16way yield similarly good results. For example, with these two designs SparseLU achieves close to 11.5x with 12 workers. For LU Picos 16way achieves slightly better results up to 8.5x. In general all three Picos designs benefit from the decreasing block sizes and scales well.

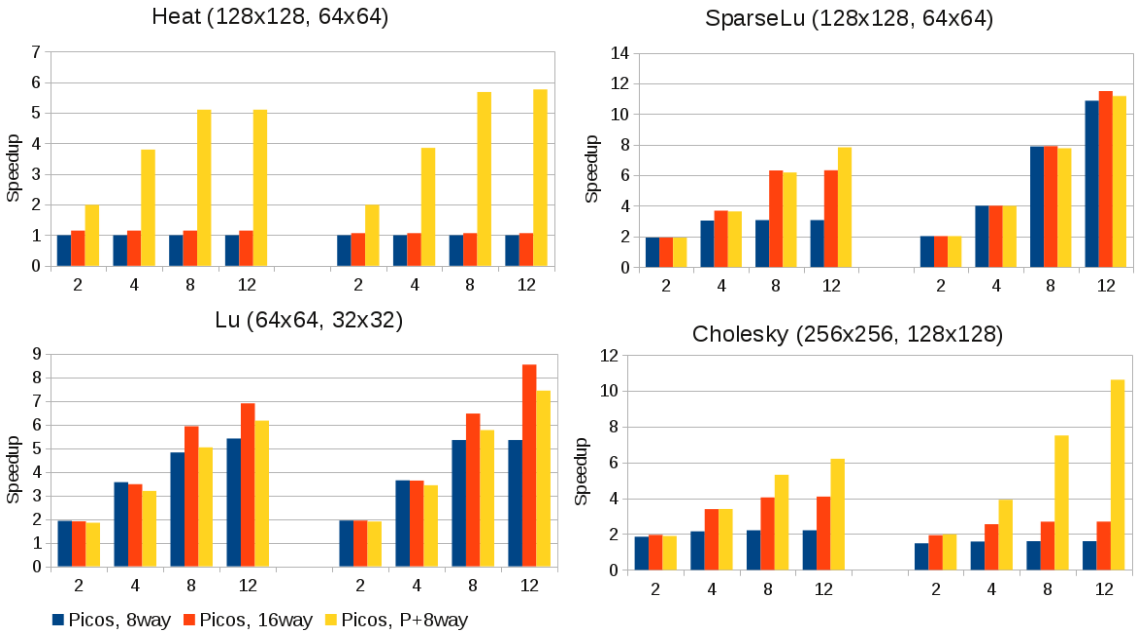


Figure 4.10: Speedup of real benchmarks by using Picos with different DM designs

Picos with DM P+8way yields better results than the other two designs in most cases. One exception is LU with DM 16way, which will be further examined later.

The performance results above are greatly dependent on the number of DM conflicts. A DM conflict is detected when a new dependence cannot be saved in any of the 8ways of the designated entry (indexed by its dependence address), because all 8 ways are occupied and are incompatible with this one. When there is a DM conflict, Picos stalls the new dependence processing until a suitable finished dependence arrives and resolves the conflict. Therefore it is essential to have less DM conflicts in the system.

Table 4.2 shows the number of DM conflicts detected during executions with 12 workers. Regarding the DM conflicts impact, we can observe that the Picos P+8way is no doubt the best solution with less DM conflicts.

Table 4.2: #DM conflicts in three Picos designs

Name	**BlockSize**	**#DM Conflicts**		
		DM 8way	**DM 16way**	**DM P+8way**
Heat	128	254	252	65
	64	1022	1020	757
SparseLU	128	189	166	0
	64	239	0	0
LU	64	491	392	0
	32	2039	1937	0
Cholesky	256	108	79	0
	128	807	792	0

The execution of LU is a corner case which is caused due to the way that Picos prototype is designed to awake the producer-consumer chain from the last consumer. With DM P+8way, there are no DM conflicts, so the task dependency graph is created much faster. When the producer task finishes, the consumer tasks are woken up from the last one, causing the schedule of some tasks in the critical path to be postponed and thus resulting in lower speedup. As for DM 16way, the task dependency graph is created much slower due to the delays caused by DM conflicts. These tasks in the critical path are therefore scheduled earlier and result in higher speedup.

To prove the cause of this behaviour and our point, we adopted two different approaches. One is to modify the task creation order of LU to avoid this corner behavior. Another is to use last-arrive-first-serve (LIFO) scheduling policy instead of the default first-arrive-first-serve (FIFO) for ready tasks. Figure 4.11 shows the performance of the modified LU (MLU) with the default scheduling and LU with a LIFO scheduling policy. As it can be seen, from block size 64 to 32, the Picos with DM P+8way now yields better results than the other approaches in both cases.

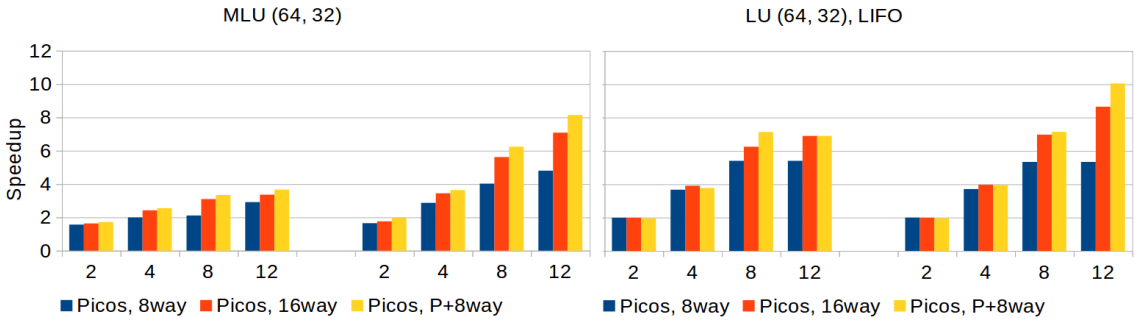


Figure 4.11: Speedup using a modified LU with the default FIFO scheduling and LU with a LIFO scheduling

4.5.2 Resource Consumption

To continue with our examination of the DM designs, Table 4.3 shows the resource consumption of the main memories of Picos for different designs, main modules and the full Picos prototypes.

The size from DM 8way to 16way is doubled with the objective to speedup this component by using higher associativity to reduce DM conflicts. This increasing can be observed in the increase in BRAM usage of DM 8way and 16way, from 9% to 17% respectively. The corresponding VM is also doubled from 512 to 1024 entries to keep it coherent

Table 4.3: Hardware Resource consumption

Design		**LUTs**	**FFs**	**BRAM(36Kb)**
XC7Z020		53,200	106,400	140
Mem	TM	0.4%	0.01%	6%
	VM for 8way/P+8way	0.4%	0.01%	1%
	VM for 16way	0.4%	0.01%	2%
	DM 8way	1.1%	0.1%	9%
	DM 16way	3.1%	0.1%	17%
	DM P+8way	1.7%	0.1%	10%
Module	TRS	1.6%	0.6%	6%
	DCT (DM 8way)	2.5%	0.3%	10%
	DCT (DM 16way)	4.1%	0.3%	19%
	DCT (DM P+8way)	2.9%	0.3%	11%
	GW+ARB+TS	1.3%	0.4%	-
Full	Picos (DM P+8way)	5.8%	1.2%	17%
	Picos (DM 8way)	5.4%	1.2%	17%
	Picos (DM 16way)	7.1%	1.2%	26%

with the DM size[87].

Resource consumption of DM 8way and P+8way are very close (BRAM usage are 9% and 10%), both are much lower than DM 16way. Although the resource consumption of DM 16way is not very demanding, its number of DM conflicts is much higher than that of the DM P+8way in Table 4.2. We could further increase the 16way into a 32way doubling the size in order to reduce the DM conflicts, but this would lead to a quadruple increase of the resource usage. Regarding the performance difference in Figure 4.10 and the hardware cost, we consider that it is unnecessary to use a DM with 32 ways. Instead Picos with DM P+8way is the most promising and balanced design among all those. Therefore, in the following sections, we will only evaluate Picos with DM P+8ways.

Hardware costs for TRS and DCT are also shown in the table, the other modules GW, TS and ARB are simple designs and their costs are small.

4.5.3 Task and Dependence Repetition Rate

In this section, the processing capacity (task and dependence repetition rate) of the hardware design is first evaluated by using the synthetic benchmarks with the HW-only mode. Then the influence of integrating the full system (ARM processing, communication and Picos) is analyzed. The cost of integrating hardware and software is mainly composed of two parts: the communication latency, the task creation and submission cost to Picos of Nanos++ runtime system (RTS). In this prototype, a bare-metal OS is used which has a much lower overhead cost than a high-level one.

Figure 4.12 shows the task creation and submission overhead measured in cycles (y-

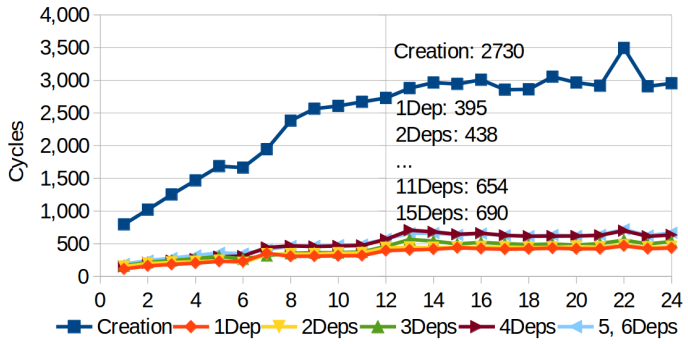


Figure 4.12: Nanos++ RTS overheads including task creation and submission cost for single task. The submission cost here only includes sending newly created task and its dependences to Picos, it does not include the computation of task dependence relations and insertion into TDG.

Table 4.4: Processing capacity of Picos P+8way

Testcase		**TestFree**			**TestChain**	**Test-1P10C**	**Test-10P1C**	**Test-10P10C**
#d1st/avg#d		0/0	1/1	15/15	1/1	2/2	11/2	11/11
HW-only	L1st	45	73	312	72	96	287	233
	rrTask	15	24	243	24	35	38	178
	rrDep	-	24	16	24	18	19	16
HW+comm.	L1st	1172	1174	1293	1151	1158	1274	1279
	rrTask	740	740	734	743	743	743	743
	rrDep	-	740	49	743	371	372	68
Full-system	L1st	3879	4240	4710	4246	4217	4531	4549
	rrTask	2729	3125	3413	3124	3168	3165	3379
	rrDep	-	3125	228	3124	1584	1583	307

axis) of the Nanos++ RTS with different number of threads (x-axis). `Creation` shows the task creation overhead per task (that is the same independently of the number of dependences); `x DEPs` shows task submission overhead from the software runtime to Picos for single task with x dependences. As explained in Section 3.1.5, when using Picos, the task submission now in the runtime only includes sending task and its dependences to Picos, without any dependence analysis. As can be seen, with 12 workers, the task creation time is 2730 cycles and the task submission cost for task with 1 dependence is 395 cycles. If an application has N tasks, the overhead of the Nth task is the accumulation of all the previous N-1 tasks.

Table 4.4 shows the processing capacity of Picos P+8way with the HW-only mode, HW+communication and Full-system modes with 12 workers. The HW-only and Full-system mode have been explained in earlier sections. The HW+communication mode basically adds communication latency based on the HW-only, and still no task creation and submission cost are considered. Row (#d1st/avg#d) indicates the number of depen-

dences for the first task and the average number of them for all the tasks. All the results are shown in 80 MHz cycles.

Firstly, in the HW-only mode, the latency of the 1st task (`L1st`) is proportional to the number of dependences of the lst task (`#d1st`). Without dependence, for example TestFree the `L1st` takes around 45 cycles. With 1 dependence, with TestFree, TestChain, it takes 73 or 72 cycles. With 2 dependences, with Test-1P10C, it takes around 96 cycles. With Test-10P1C to Test-10P10C with 11 dependences for the 1st task, `L1st` take similar time as 287 or 233 cycles. Finally with TestFree with 15 dependences per task, `L1st` takes 312 cycles.

Secondly, the task repetition rate (`rrTask`) mainly depends on the average number of dependences (`avg#d`) and the type of the dependences. With 1 dependence per task in average with TestFree and TestChain, `rrTask` costs 24 cycles. With 2 dependences per task in average with Test-1P10C and Test-10P1C, although they have different dependence patterns, their `rrTask` take 35 or 38 cycles. With 11 dependences per task in average with Test-10P10C, it costs 178 cycles. With 15 dependences per task with TestFree, it costs around 243 cycles.

Thirdly, the dependence repetition rate (`rrDep`) remains stable in all the test cases and decreases when the average number of dependences (`avg#d`) increases. For benchmarks with only one dependence, the `rrDep` is around 24 cycles. For benchmarks with more dependences, it is around 16 to 19 cycles. This can be seen as the repetition rate of the first and following independent instructions flow into a pipelined functional unit.

However, this effect for `L1st` and for `rrTask` in the HW-only mode is nearly hidden in the HW+comm and the Full-system modes, showing that the hardware part is fast enough with the current integration. Moreover, the effect for `rrDep` in the HW-only is greatly enhanced in the Full-system mode where the communication and Nanos overheads become the main performance factor. As can be seen, for the HW+communication to the Full-system mode, the `rrDep` with TestFree from 1 to 15 depenences per task drops from 740 to 49 cycles and from 3125 to 228 cycles, respectively. Indeed, in the Full-system mode, as the number of dependences increases, the time required to process a task `rrTask` remains stable while the `rrDep` decreases proportionally. This is a key factor contributing to the powerful performance of Picos with the Full-system mode presented, as this effect does not appear in the software-only implementation.

4.5.4 Scalability

Finally to show the real potential of the Picos prototype, real benchmarks are used for its scalability studies with up to 24 workers. Figure 4.13a to 4.13e show the speedup (bar, y-axis) of Heat, SparseLU, LU, Cholesky and H264dec with four block sizes obtained by Picos with the Full-system mode, the Perfect simulator and the Nanos++ RTS (software-only implementation).

Results of the Perfect simulator shows the available peak parallelism in these applications. When compared with this perfect speedup, with Heat, SparseLU, LU and Cholesky, the Picos prototype achieves nearly roofline speedup with block sizes from 256 to 64 and with up to 24 workers. With H264dec, it scales very well with block size (8, 4, 2, 1) and up to 12 workers, then remains stable.

With Heat, Cholesky with block size 32 and H264dec, there are emerging gaps between the results obtained by the Picos prototype and the Perfect simulator. There are several reasons here, firstly the Perfect simulator calculates an ideal speedup of applications without any communication and OS costs. Secondly, the first prototype is the simplest configuration with only one TRS and DCT, which is unable to unfold such a high and fine parallelism from applications here. For example, with H264dec with 1x1, there are 139934 tasks with the average execution time of 39400 cycles per task in Table 4.1. With this simple configuration, it also lacks hardware resources to manage so many processors. Notwithstanding, the Picos prototype with more module instances should be able to obtain higher speedup and fill this gap[87].

Compared to the Nanos++ software-only RTS, Picos greatly exceeds its performance in all the benchmarks. For each benchmark, given a fixed block size, Nanos++ RTS scales up to 8 workers maximum while the Picos prototype continues to scale up to 24 workers in some cases. For example, for SparseLU with block size 32 and Cholesky with block size 64, the Picos prototype achieves 16x to 24x and 15x to 21x with 16 and 24 workers, respectively.

More importantly, for each benchmark, as its block size decreases, Nanos++ RTS starts to degrade rapidly after some point while the Picos prototype keeps on advancing or at least remains stable. For example, in Figure 4.13a, the speedup achieved by Nanos++ RTS for Heat (64 to 32) drops from 4.5x to 1.6x with 8 workers while that by Picos prototype remains stable as 6.3x; and for SparseLU in Figure 4.13b and LU in Figure 4.13c with block size 64 to 32, when the speedup achieved by Nanos++ RTS starts

to degrade, the Picos prototype continues to advance from 3.3x to 8x with 16 workers. In Figure 4.13e, for H264dec, as the block size and the performance of Nanos++ RTS decreases, Picos prototype remains stable.

All of the above prove that even the simplest configuration of Picos Hardware fulfills our expectations, and larger configurations are expected to be able to cope with future many-cores.

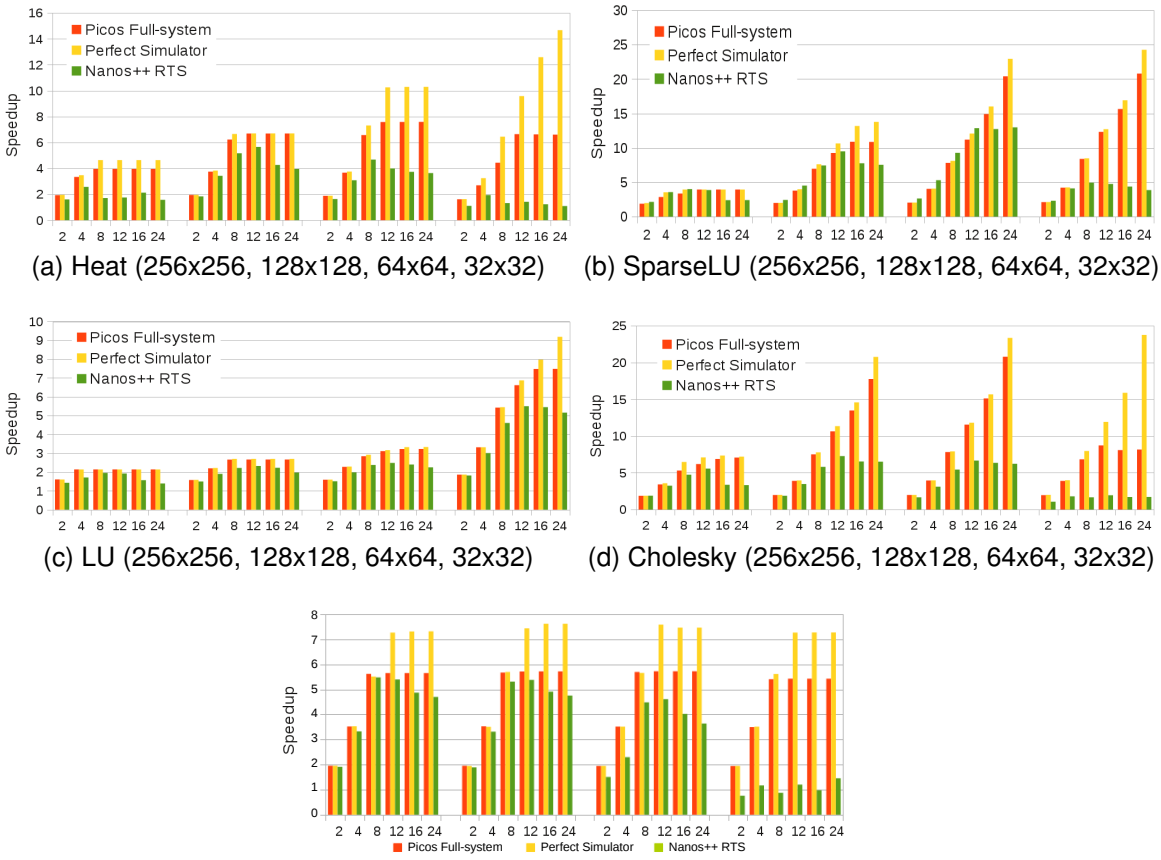


Figure 4.13: Strong Scalability (Speedup) results of Picos Full System, Perfect Simulator and Nanos++ RTS (software-only) for applications with different block sizes, compared to the sequential version.

4.6 Summary and Concluding Remarks

Task-based programming paradigms such as OpenMP 4.5 and OmpSs are ubiquitous approaches to program multicore and many-core architectures. They are simple to use, and are powerful to gain high performance with coarse-grained tasks. However, their

software runtime overhead especially the task dependency graph management and thread contention prevent the exploitation of fine-grained parallelism, which leads to performance degradation among applications.

In order to avoid this problem and extend the benefit of these programming models into a finer-grained parallelism, this chapter proposes and presents the very first hardware prototype of Picos, as a hardware task dependence manager. The presented implementation is coded in VHDL, has been carefully analyzed and tested on an embedded system on a Zynq 7000 SoC Platform (Zedboard).

This prototype is able to manage 256 in-flight tasks with up to 15 dependences per task. This configuration was selected based on a previous work which used a C simulator to perform design space exploration [87]. In this previous work, a design like this prototype is estimated to be able to scale up to 8 workers while a larger design with four times the size is able to scale up to 256 workers without damaging the performance. With real benchmarks, our prototype Picos is able to scale up to 24 workers, which has greatly exceeded the estimation from the design space exploration. In addition, the performance results have also been compared with a State-of-the-Art software-only runtime system, and Picos greatly outperforms the software-only alternative with fine-grained tasks.

This first prototype is a milestone, as it sets the foundation architecture for all our future designs, and it realized the ideas in our mind into a real hardware.

Chapter 5

Nested Task Suppport -Second prototype

Standard task-based programming models such as OpenMP and OmpSs define nested tasks as tasks that have been created by other tasks, with or without dependences. The OpenMP and OmpSs standard define two limitations for nested tasks, which are necessary to consider:

1 A "child task" can only have a subset of the set of dependences of its parent task.

2 The dependences of a "child task" only affect its sibling tasks ("child tasks of the same parent").

Nested tasks are generally supported to improve the programmability of parallel programs. For instance, a nested task can be found in recursive codes where the recursive function is a big task and can be further decomposed into smaller tasks for more parallelism. Another usual circumstance for the use of nested tasks is the case where a task is used to call a library that has already been programmed with tasks. Therefore, nested task support is a necessary feature of any task manager that wants to execute general-purpose codes.

Although our first prototype Picos, presented in previous chapter, obtains much higher performance than the software-only runtime, it does not support nested tasks in hardware. This fact reduces the programmability and exploitation of parallelism as for example the case of task-based recursive functions or task-based libraries.

The main reason why Picos does not support nested tasks is due to the fact that hardware task managers, opposed to software ones, have a limited amount of memory. Thus it is possible that a hardware task manager reads a task and runs out of memory space in the middle of processing task dependences. Without nested tasks, this is not a problem because previously processed tasks are scheduled to execute when they are ready and eventually they will finish. When these previous tasks finish their execution, they free

resources that allow the hardware to proceed with the processing of the stalled task. For nested tasks with dependences, the situation is much more complicated and can lead to system deadlocks.

This chapter describes the second prototype Picos++ [72], which proposes a hardware/software co-design to extend the first prototype of Picos with the support for nested tasks. The first condition to avoid deadlocks is to read and process tasks atomically. Once a task is read, all its dependences should be processed (i.e. integrated in the task dependency graph). Additional actions are required to deal with the task and the remaining dependences when the hardware is used up. In addition, actions are required to resume the hardware processing when the full situation is cleared.

The main contributions of this chapter can be summarized as follows:

- A hardware/software co-design for supporting nested tasks in Picos++. In the hardware part, new information and an additional memory structure have been added to the design. In the software part, a control system has been added to prevent deadlocks when the hardware resources are fully used or when internal memory conflicts appear in Picos++.
- A different communication mechanism between the General Purpose Processor and the hardware accelerator.
- A fully integrated Picos++ system based on a Xilinx Zynq 7000 series SoC with Picos++ connected with the processors in hardware. This includes the integration of Picos++ with a State-of-the-Art programming model and a Linux OS in software and consequently the capability of executing real applications using Picos.
- Performance and energy consumption analysis of applications executing in real hardware when running a task-based dataflow programming model using a software-only compared with a Picos++ accelerated runtime.
- Visualization of real application executions and Picos++ activities.

5.1 Picos++ system

Picos++ is an evolution of the Picos design. In order to support nested tasks without deadlocks, a hardware/software co-design is introduced in Picos++. In the hardware part, additional hardware support is added to ensure atomic task processing. In the software

part, a mechanism named buffered task recovery is introduced to intervene when it detects a potential deadlock threat during execution. Additionally, the data communication is also modified to allow higher performance and easier support for the co-design. This section shows a conceptual view of Picos++ system, and explains the different data communication mechanisms used in both Picos and Picos++.

5.1.1 System Organization

Figure 5.1 shows the Picos++ system based on a commodity SoC. It consists of the Processing System (with 2 ARM cores), the Programmable Logic, the main memory and other peripherals required for the system to work. Applications, programming models and Picos++ APIs reside in the Processing System. Picos++ and its communication logic reside in the Programmable Logic.

The Processing System and Programmable Logic are connected to the main memory where they share three buffers managed as circular FIFO queues. Each buffer stores up to N units (each unit represents one task). Each unit inside the new task buffer consists of these fields: *TaskID (8 bytes), Number of Dependences (4 bytes) and Dependence Memory Address and Direction (12 bytes) of each dependence for up to 15 dependences*, in total 192 bytes. Each ready task unit includes *TaskID, Picos++ID (4 bytes)*. Each finished task is represented by its *Picos++ID*.

Special values of *TaskID* and *Picos++ID* in the new and finished task unit are reserved in order to indicate a valid task. Also, a special value of *Picos++ID* in the ready task unit is used to indicate empty space in this unit.

5.1.2 Number of Dependences per Task

One of the fixed parameters of the hardware is the maximum number of dependences per task, fixed to 15 originally as a "more than enough" value. Although currently all the evaluated applications have less than 15 dependences and thus fit in the current system, it is possible that this changes in the future with new applications.

To solve this problem a hardware-software mechanism has been designed that allows Picos++ to process tasks with any number of dependences. As the number of dependences is known at compile time, if any corner case with more dependences appears, the software part of the runtime will create two (or more if necessary) helper tasks to accommodate them. Figure 5.2 shows how an original task A with 16 dependences is splitting into three

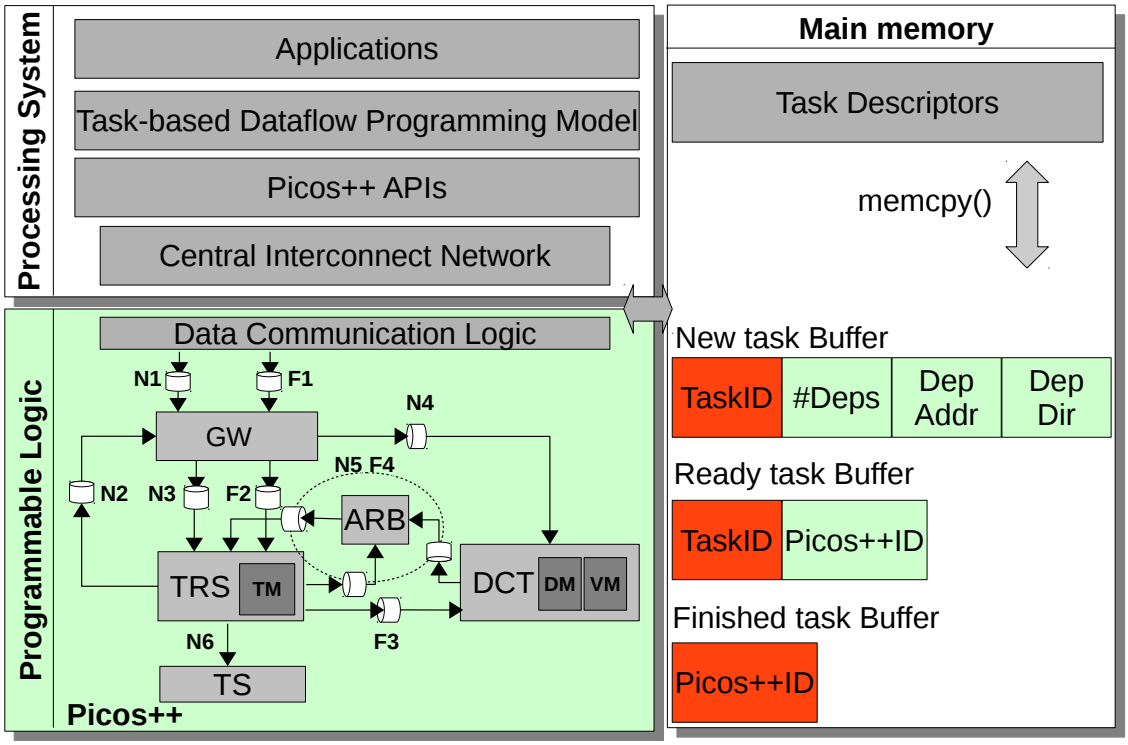


Figure 5.1: Picos++ based on Zynq 7000 SoC

tasks A1, A2 and A3 that have 15 or less dependences per task and maintains the correct execution order. Task A1 will be created with 14 of the original dependences and an Extra (E) helper dependence with inout type. It will be a empty task that has no execution time. Task A2 will have the remaining 15th and 16th original dependences and the same Extra (E) helper dependence with inout type. It will inherit all the computation from the original Task A. This way through the inout dependence chain, the original code will be executed only after all the original dependences are ready. Finally, another empty helper task A3 will be the same as task A1. This last task will ensure that any task that depends on any of the 16 original dependences is executed in the correct order. This simple mechanism can be replicated for any number of dependences using more helper tasks. In fact, it allows Picos++ to process tasks with more dependences than fit in its internal memories (as the helper tasks don't need to be processed as a single unit) and maintains the parallel processing in a nested task environment.

5.1.3 Data Communication

The data communication between the general purpose processors (GPPs, the ARM cores) and the hardware task manager is one of the key factors that determines the final system

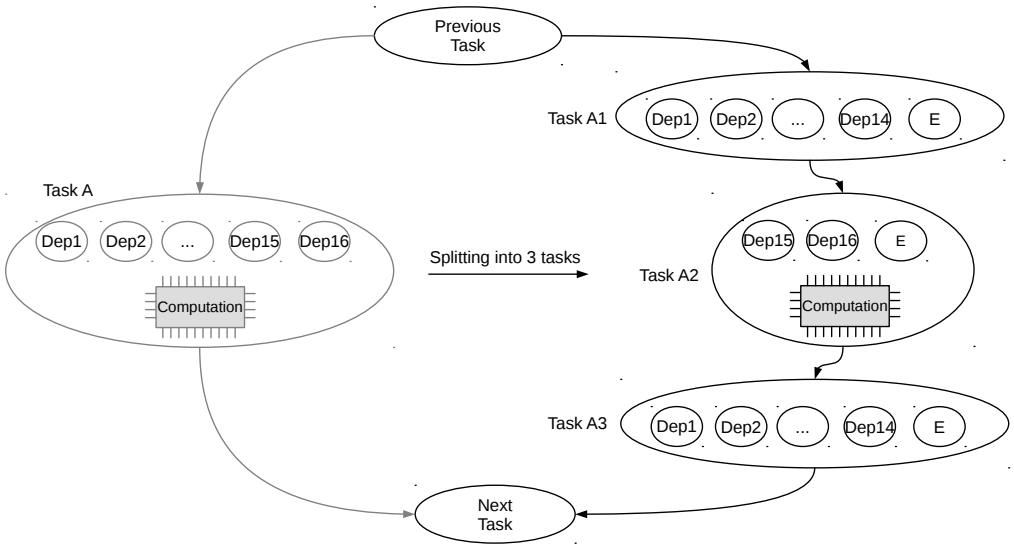


Figure 5.2: Software mechanism to tackle tasks with more than 15 dependences

performance. The first prototype Picos uses synchronous communication. As shown in Figure 5.3a, the threads running inside GPPs are in charge of programming DMA for each data movement (new, ready, finished task) from memory to Picos or backwards. In this case, Picos is a slave module, which relies on DMA for all the data transfers. Although simple, this method is costly due to the frequent small data exchange between Picos and threads, and the constant interference required from GPPs. Although DMA is also useful for asynchronous communication, its benefit is not obvious due to the frequent small data exchange pattern in our user case. As a result, the overall system performance is burdened.

Picos++ implements a buffered and asynchronous communication which requires much less support from GPPs as shown in Figure 5.3b. During the initialization process, the threads running inside GPPs allocate three buffers (new, ready, finished) in the memory and send the addresses and lengths of these buffers to Picos++. Picos++ has a master module which can then access these buffers in memory without further intervention from the threads. It decouples the data communication between software and hardware, ensuring a more efficient small data exchange and releasing GPPs for more useful work. As a result, there is a significantly higher overall system performance.

The operational flow of the communication module in Figure 5.1 follows: 1) **Read new task**. It reads the *TaskID* field of a unit in the new task buffer. If this is valid, it reads the new task and then invalidates this unit. Otherwise, it tries to write a ready task. 2) **Write ready task**. It reads the *TaskID* field of a unit in the ready task buffer. If this unit is empty and Picos++ has a ready task, it writes the ready task from Picos++ to the ready

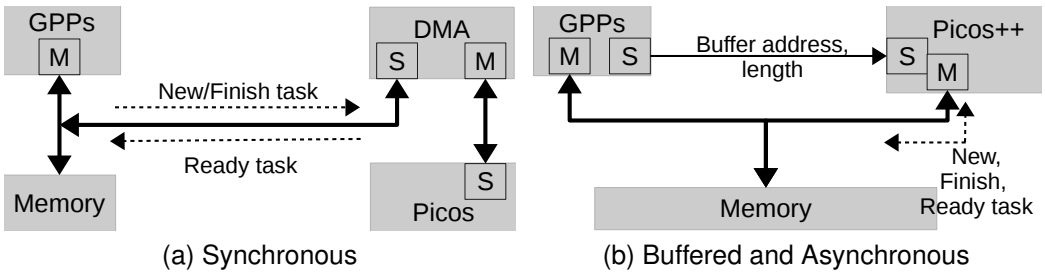


Figure 5.3: Data communication between Picos designs and General purpose processors

task buffer. Otherwise it checks for finished tasks. 3) **Read finished task**. It reads the *Picos++ID* field of a unit in the finished task buffer. If this is valid, it reads the finished task and afterwards invalidates this unit. Otherwise, it checks for new tasks. This process starts when software writes a start to the data communication logic, and stops when it writes a stop to the communication logic.

Certain modifications are required in the software runtime in order to offload the dependence analysis to hardware and to support the hardware/software co-design for nested tasks. The main operations of the modified software runtime are as follows: (1) It opens Picos++ for access and allocates three buffers for new, ready and finished tasks in DDR3 (as shown in Figure 5.1). Afterwards, it sends the buffer addresses and lengths to Picos++ and starts the following process. (2) It creates tasks and copies them to the free spaces in the new task buffer. (3) It checks for ready tasks in the ready task buffer and copies them to the ready task pool for worker threads. Afterwards it resets the entries in the ready task buffer to an empty state. (4) When there are finished tasks, it copies them to the free space in the finished task buffer. (5) When the process finishes, it deallocates buffers and stops Picos++.

5.2 Nested task and Software support

5.2.1 Deadlock Scenario

In order to support nested tasks it is necessary to have an additional field in the new task packet containing the parent ID of the task. Picos++ reads this field and sends it to the DCT with every task dependence. After that, the corresponding DCT differentiates dependences using the parent ID as an additional tag. This extension tag allows the dependences of non-sibling tasks to be maintained in separate task dependency graphs, so inter-task dependence relationships are only applied between sibling tasks (due to the

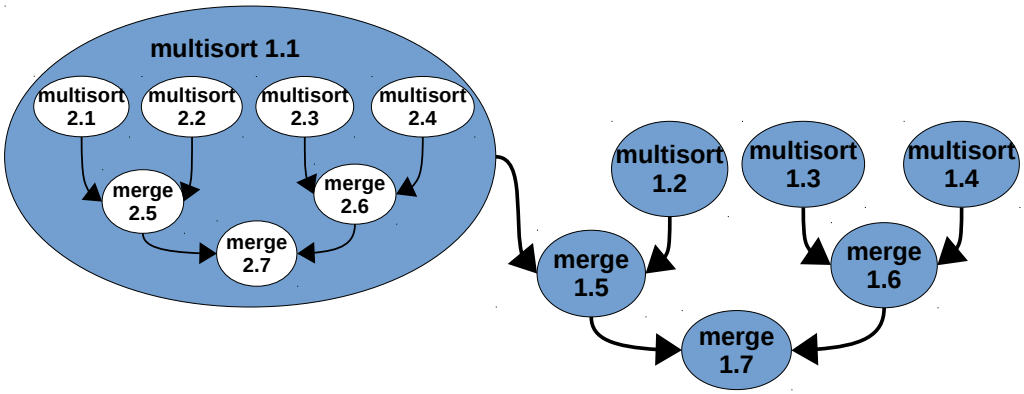


Figure 5.4: Multisort with nested task dependences

second limiation of the definition of nested tasks mentioned earlier).

Nonetheless, the above implementation is not enough to avoid corner cases that lead to a whole system deadlock. Those corner cases derive from the fact that hardware task managers, opposed to the software ones, have a limited amount of memory. Thus it is possible that hardware task dependence managers read a task and run out of internal memory space in the middle of processing task dependences. Without nested tasks, this is not a problem because previous processed tasks go to execution and eventually they finish. and free out resources that allow the hardware to proceed with the stalled task.

For nested tasks with dependences, the situation is much more complicated. As an example, Figure 5.4 shows two levels of nested tasks, where tasks in level `i` are labeled `i.X`. While all tasks in level 1 are shown, only child tasks of multisort 1.1 are shown for the second level.

To simplify the deadlock scenario, assume that Picos++ has memory capacity for only 7 tasks. In this case, the first level tasks would quickly fill the Picos++ manager before the first task multisort 1.1 goes to execution and starts creating new tasks. In particular, at the moment that multisort 1.1 creates the first child task multisort 2.1, this task will go into the hardware manager and gets stuck due to a full memory problem. As the parent task multisort 1.1 will have to wait until all its child tasks end (and they will not end), it will not able to finish and all the following multisort and merge `1.X` tasks, already inside the hardware manager, will not able to proceed (and free memory for more tasks), so the system will go into a deadlock.

5.2.2 Deadlock Free Hardware/Software Co-design

In order to support nested tasks without deadlocks, we introduce a hardware/software co-design in Picos++. The explanation of this co-design and how it works follows.

5.2.2.1 Atomic Task Processing

The first condition to avoid deadlocks is to read and process tasks atomically. Once a task is read all its dependences should be processed (i.e. integrated in the task dependency graph). The first prototype Picos uses a non-atomic task processing, where until the FIFO queues connecting GW with TRS and DCT are full, GW can read new tasks without considering the free spaces in all Picos memories (TM, DM and VM). Without nested tasks, this method ensures efficient inter-component communication. However, it leads to deadlocks easily with nested tasks. For instance, assume the scenario shown in Figure 5.4 where task multisort 2.1 is stuck in those FIFO queues. Thus, in Picos++ the hardware is modified to read and process tasks as a whole, which means that the GW is conscious of the free spaces in all Picos++ memories before it reads a new task.

This awareness has two cases to consider: when the new task has no unique dependences (all of them are versions of dependences already existed in the DM and VM in the hardware) and when there are unique dependences. For the first case, it is easy to ensure the processing of a new task and new versions of already known dependences as they can be stored in any empty entry in their respective memories (TM and VM). In this case, one empty space in the TM and 15 empty spaces in the VM are enough to read a new task. The implementation of TM and VM allow to know the amount of their empty spaces in advance easily.

For the second case, ensuring the processing of new dependences is more difficult as they are stored using a hash process in order to have a fast location mechanism in DM [71]. DM uses hashing and a 8-way associative memory, a full-associative memory has been discarded due to its complexity and resource requirements. Therefore, a dependence may not be stored because all the 8 ways of the DM entry indexed by the dependence address are full, independently of the whole DM usage. This problem can not be anticipated and results in a negligible slow-down without nested tasks, but with nested tasks may lead to a deadlock.

To overcome it, Picos++ has a new fall-back memory that is able to store up to 16 dependences. This memory is used whenever a new dependence cannot be stored in DM.

Once it is used, it raises the memory full signal. This signal stops the GW from reading more new tasks while allows it to continue processing the remaining dependences of the current task. As soon as DM has free space, the dependences are moved from the fall-back memory back to DM, so Picos++ resumes new task processing.

5.2.2.2 Buffered Task Recovery

Atomic task processing alone cannot avoid deadlocks. If the aforementioned child task multisort 2.1 is stored inside the new task buffer, but cannot be read by the hardware, the system stalls. This is only an issue for the first non-executed child of its parent. Due to the fact that child dependences are always a subset of their parent, the first child is always ready. This does not hold true for the second and subsequent children. Therefore, if the first child is still in the new task buffer, it has to be processed (somehow) in order to avoid deadlocks. Otherwise (if the first child is not stored in the new task buffer) there is not going to be a deadlock. Thus, the remaining children of a task must remain in the queue until the hardware processes them in order to avoid race conditions.

To fulfill this requirement, the software support of Picos++ keeps track of the entries in the new task buffer where a parent has stored its children. Whenever a full condition arises, it checks the state of the first child. If the first child task has been read, the software support stays at normal routine (the main operations). Otherwise the software support intervenes to avoid a possible deadlock. First, the thread locks the new task buffer and removes all its child tasks. Then, the buffer is reconstructed if it has any remaining tasks (created by other running tasks) by updating all the corresponding pointers. Afterwards, either the child tasks are directly executed in order by the thread (without allowing them to create more tasks) or submitted as a whole to a software task dependency graph manager. Finally, when the full condition in hardware is cleared, the software support reverts to use hardware for dependence processing. The first option is simpler and keeps the complexity of the software part at bay. The second may allow extra parallelism to be extracted in some corner cases.

5.2.3 DM and Fall-back Memory Design

As we mentioned in the previous chapter, DCT is the major dependence management unit. It manages task dependences through one Dependence Memory and one Version Memory (DM and VM respectively). For each new arrived dependence, DM performs address

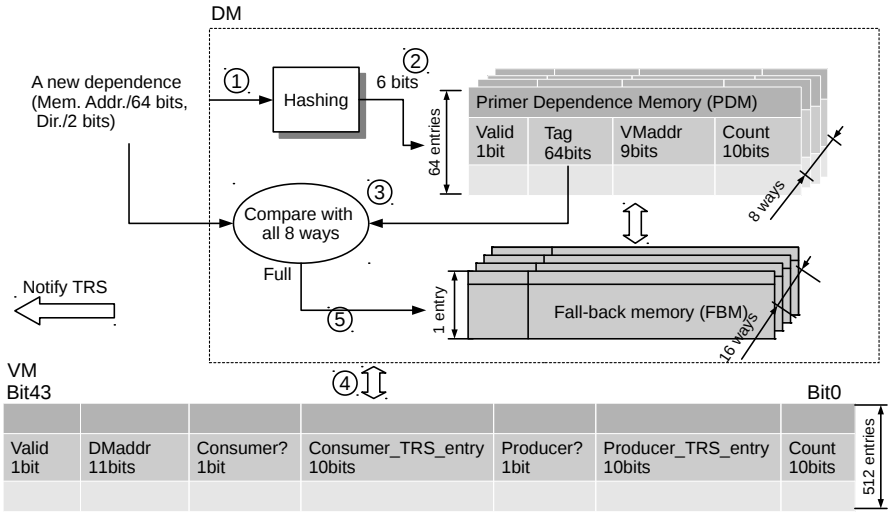


Figure 5.5: DCT Memories organization

match of it to the tags of all these arrived earlier. The dependence with unique memory address is allocated with a new space in DM and is saved as a tag, otherwise DM saves it in the matched place belonging to the earlier arrived dependence. VM receives instructions from DM and keeps the chain of consumers and producers of all the references to the same dependence address (called versions).

In order to support nested tasks in hardware, a fall-back memory as part of a new DM design is introduced to ensure space in DM for one new task. Figure 5.5 shows this new DM design in the second prototype. It includes both the primary dependence memory (PDM) (the same design as the original DM in the first prototype), and the fall-back memory (FBM). The PDM is a 64-entry, 8-way associative cache-like memory while the FBM is a 1-entry, 16-way associative design.

Figure 5.5 also shows VM in this prototype. VM is a 512 entry memory, implemented by using a BRAM with 512 depth, it accepts free entry requests and recycles used entries. When compared to the VM design in the first prototype, the only difference here is the width of the DM address (11bits, we will introduce it later) stored inside.

The new DM operational flow can be seen in Figure 5.5. When a new dependence packet enters the DM memory (circle 1), first all the 64 bits of its memory address go through the hashing module to obtain a 6 bits entry address (circle 2). Then the 64 bits are compared against all the `tags` from the 8 ways (circle 3). Then depending on whether there is a hit or miss, the direction of this dependence, and the usage of the 8 ways of the PDM entry, DCT performs different actions to update PDM and VM, and sends different notifications to TRS. This has been documented in great detail in Chapter 4 and therefore

it is not repeated here. However, one outcome is processed differently in this new DM design. When there is no hit and all the 8 ways of the PDM entry are full, instead of stopping DCT to process this new dependence, this new dependence goes to the FBM (circle 5). When the FBM is used, GW will stop reading new tasks. Therefore there are two possible situations in Picos++. First, if this is not the last dependence of the current new task, it is guaranteed that the current new task is the only new task in Picos++, therefore 14 more spots are the maximum number required to save it. Second, if it is the last dependence of the current new task, then potentially there can be another new task in Picos++, in this case an additional 15 spots are required. With 16 ways, FBM is able to handle both situations. VM is updated accordingly and the corresponding communication dependence packets are sent.

When a finished dependence packet enters, the associated VM address is first used to read the `Consumer` information and the `DMaddr` in the VM entry. If there is a consumer task, it sends a ready notification to TRS with the `Consumer_TRS_entry` to wake up the last consumer task in chain, and the `count` in both VM and DM are deducted by 1; afterwards, if the `count` value reaches 0, and the VM entry has a `Producer` information, a ready notification is sent to TRS. When TRS receives a ready notification, it checks if the task is ready-to-execute and also TRS in turn wakes up the previous consumer task if it exists. When each of these consumer tasks finished execution and their finished dependence packets enter, the same procedure continues. When the `count` reaches 0, the corresponding VM entry or DM way is freed.

Since Picos++ supports tasks with up to 15 dependences, we mentioned earlier that the FBM has the ability to save all the dependences from the tasks already read. Whenever the FBM is used, the GW stops reading new created tasks but the processing of finished tasks continues. Once the suitable space (one of the ways in the corresponding entry) is emptied in the PDM, the context in FBM is moved to that entry in PDM. Note that, the context in FBM cannot be moved back to a random way in a random entry, it has to be one of the ways in the same entry after hashing, therefore FBM has to remember this entry address.

The PDM is implemented by using 8 BRAMs with 64 entries each, therefore to access a specific way, 9 bits address (6 bits for entry, 3 bits for way) is required. The FBM is realized by using 16 registers and the necessary control logic, a 4 bits address is required to access one of its 16 ways. To distinguish these two memories in the new DM design, and for the movement of content from FBM to PDM, we extend the address to access

DM from 9 bits to 11 bits. If Bit10 of the address is zero, then Bit8 to Bit0 form the 9bits address needed to access the PDM; otherwise, the FBM is accessed. In addition, Bit9 to Bit4 inherited the entry address that can be used to move between PDM and FBM, and Bit3 to Bit0 are used to access the FBM. This DM design works correctly with applications with or without nested tasks, it is fast and has a low hardware resource consumption.

DCT is a complicated module for several reasons. On one hand, dependence address matching gets very complicated and time consuming as the new tasks are created much faster and in great number when exploiting fine-grained parallelism. This effect is responsible for the performance degradation in the software-only runtime. On the other hand, it is desirable to have a balanced speed between different design parts, such as TRS and DCT, to achieve a good performance and be cost wise as a chain is only as strong as its weakest link. However, this is not easy to achieve because each task can have up to 15 dependences. That fact stresses DCT much more than TRS. Additionally memory addresses of dependences always tend to group in clusters for many applications, leading to a lot of hot spots in memory, and system stalls. Finally, with nested tasks, the fact that implementations of hardware task managers, opposed to the software ones, have a limited amount of memory can lead to corner cases that result in a whole system deadlock. All the above reasons are considered for the design of DCT and even the whole prototype.

5.3 Experimental Setup and Benchmarks

5.3.1 Experimental Setup

Picos++ is coded in VHDL and implemented in a Zynq 7000 series SoC platform (Zedboard). Its communication logic is coded in C with Vivado HLS directives. For the experiments in this chapter, Picos++ and its communication mechanism were synthesized with Xilinx Vivado Design Suite 14.4. Zedboard includes one FPGA Chip XC7Z020-CLG484 [85] which comprises a Processing System (PS) with 2 ARM Cortex-A9 (working at 667MHz) and a Programmable Logic (PL) part (at 100MHz).

The evaluation of Picos++ is done with OmpSs[35], a forerunner for the OpenMP Standard. OmpSs is supported by the source-to-source Mercurium compiler and the Nanos++ runtime system. Picos++ uses a modified version of Nanos++, as its own software counterpart runtime. We also use performance tools Extrae and Paraver[26]

to analyze the application behavior in our system. Sequential and parallel execution time of OmpSs applications are obtained in real executions in Zedboard which operates on Ubuntu Linaro Linux 14.04.

5.3.2 Benchmarks

A set of synthetic and real benchmarks[25, 79] are selected to show the capability of Picos++, the impact of different communication systems and nested task support.

Synthetic benchmarks include TestFree, TestChain and TestNested. TestFree consists of tasks that are all independent with each other, thus is selected to test the maximum ability of Picos++. TestChain is the opposite, it consists of tasks with dependences that form an inout chain, thus there is no paralellelism at all. It is selected to show the worst case of Picos++. TestNested is selected to test the functionality of nested task support in Picos++.

Real benchmarks include Cholesky, Multisort and H264dec. Cholesky is a representative application in scientific computing with complex dependence patterns. Multisort is a representative application with a recursive algorithm. Finally, H264dec is a popular media/video decoding application. Both Multisort and H264dec are chosen to test not only the task dependence management, but also the nested task support. Their functionality description can be found in Chapter 3, Section 3.2.

Table 5.1: Characteristics of Real benchmarks

Name	Configurations	#Tasks	AveDep	AveTSize (ns)
Cholesky	2k 128	816	2.5	6074213
	2k 64	5984		887365
	2k 32	45760		128693
	2k 16	357760		22239
	2k 8	2829056		4627
H264dec	10f 16 16	549	6	3138500
	10f 8 8	1580		998445
	10f 4 4	5490		289683
	10f 2 2	21110		89809
	10f 1 1	82310		32293
Multisort	128k 4k 32k	45	2	1994309
	128k 1k 32k	157		639705
	128k 256 32k	605		237811
	128k 64 32k	2397		219350

Table 5.1 shows the characteristics of the real benchmarks with different problem and task sizes. Column configuration shows the problem and block size for Cholesky; the number of frames and block sizes for H264dec; the input array size, the minimal sort size

Table 5.2: Hardware Resource and Power Consumption

Name	Resource			Power
Design	LUTs	FFs	BRAM(36Kb)	Watts
XC7Z020	53,200	106,400	140	-
Picos++	11.32%	4.22%	18.57%	0.011
Data Communication Logic	3.90%	2.93%	0%	0.009
2 ARM Cortex-A9 cores	-	-	-	1.53

and the minimal merge size for Multisort. Column #Tasks and AveDep shows the number of tasks and average number of dependences per task. Finally column AveTSize shows the average task size in nanoseconds of these real applications.

5.4 Results

This section examines the Picos++ system in both performance and energy consumption. It starts with an assessment of the hardware and power cost of Picos++. Followed with a comparison between Picos++ with the first prototype Picos regarding their different communication schemes and the influence on their processing capacity. Afterwards performance and energy studies of Picos++ are presented by using both synthetic and real benchmarks. Then a visualized analysis of application execution in the Picos++ system is illustrated to show insights for higher energy efficient designs in the future. Finally since the commodity hardware used for building Picos++ system has a limited 2 threads, a discussion of Picos++ scalability with more threads and resources is presented.

5.4.1 Hardware Resource and Power Consumption

Table 5.2 shows the on-chip resource utilization and dynamic power consumption of Picos++ on the Xilinx XC7Z020-CLG484 chip [85]. All the results are obtained from Vivado post-implementation reports. The on-chip static power (not shown in the table) is 0.163W. The Processing System which includes 2 ARM Cortex-A9 CPUs takes around 90% of the on-chip power consumption. Picos++ and its data communication logic consume a small fraction, around 0.02W, less than 1.3% of the chip power. This, and the fact that Picos++ requires less than 20% of the Programmable Logic, makes it feasible to be integrated in multicore CPUs. Since the design of Picos++ is stable, its hardware and power consumption will be similar if implemented in other different FPGAs.

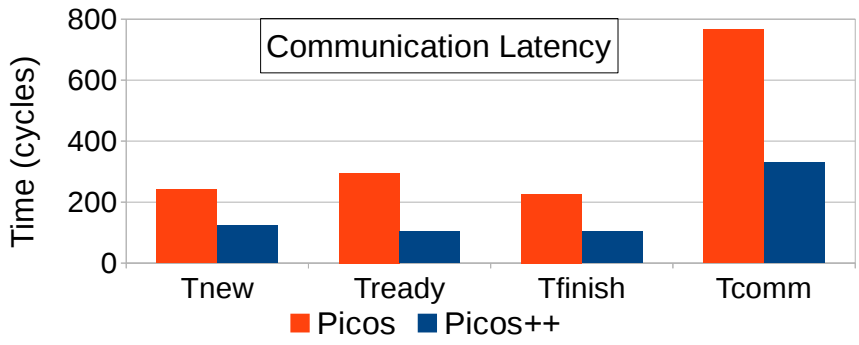


Figure 5.6: Communication latency in Picos and Picos++

5.4.2 Picos++ versus Picos

In this section, Picos and Picos++ will be compared regarding their communication latencies, task and dependence repetition rates, and performance.

5.4.2.1 Synchronous versus Asynchronous Communication

Figure 5.6 shows the communication latencies shown in cycles (at a frequency of 100MHz) between the CPUs and Picos or Picos++. Tnew, Tready and Tfinish are the latencies for transferring a new, ready and finished task message. Tcomm is the accumulation of all three previous latencies. The communication latency is similar for tasks with different number of dependences. As it can be seen, for one task the communication in Picos++ is more than 2x faster than in Picos.

5.4.2.2 Task and Dependence Repetition Rate

Table 5.3 shows the task and dependence repetition rates (`rrTask` and `rrDep`) of Picos and Picos++ by using TestFree and TestChain with different number of dependences, and with empty tasks (0 execution time). The results are shown in 100MHz clock cycles with 2 threads. Note that the Picos results vary from Table 4.4 for the Full-system since the latency in cycles now accounts the real execution of the application with real software task creation and Linux OS overheads, when previous chapter shows results with simulated task creation and execution, in addition without Linux OS.

HW-only indicates results without any communication latencies or software runtime overheads, and the evaluation results were based on execution traces. The Picos and Picos++ rows show results obtained in real executions (both are integrated with a modified version of the Nanos++ runtime where the dependence analysis is offloaded to the hard-

Table 5.3: Task and Dependence Repetition Rates

Testcase		**TestFree**		**TestChain**	
Average number of dependences		1	15	1	15
HW-only	rrTask	24	243	35	348
	rrDep	24	16	35	23
Picos	rrTask	8418	9060	9359	10424
	rrDep	8418	604	9359	695
Picos++	rrTask	1288	1635	1771	1791
	rrDep	1288	109	1771	119

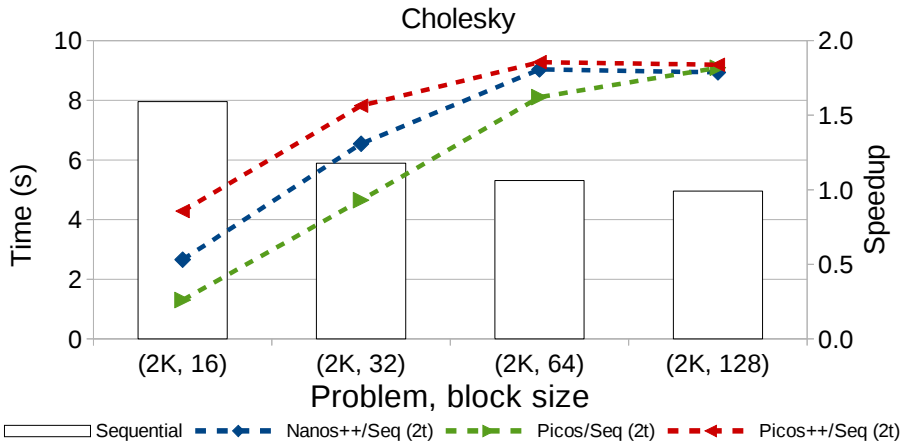


Figure 5.7: Execution time and speedup of Cholesky by using Picos and Picos++ with 2 threads

ware).

Picos and Picos++ have similar results for HW-only, thus we only show one set of results for both. In this scenario, the task and dependence repetition rates are really small. The task repetition rate is proportional to the number of average dependences and it increases slightly with a more complex dependence pattern. For example, for tasks with 1 dependence in TestFree and TestChain, it takes around 24 and 35 cycles respectively. With 15 dependences, it takes around 243 to 348 cycles. The dependence repetition rate is quite steady for different dependence patterns from 16 cycles to 23 cycles. This is due to the fact that Picos and Picos++ pipeline the processing of all the dependences of a task.

The data communication mechanism and the integration of Picos++ may have significant impact over system performance. In Picos, the communication cost is too high to gain any performance for real benchmarks.

Figure 5.7 shows the performance results for Cholesky by using Picos, Picos++ and the software-only Nanos++ runtime. The application uses a matrix size of 2048x2048 and different block sizes from 16x16 up to 128x128. The primary Y-axis shows the se-

quential execution time in seconds in bars. The secondary Y-axis indicates the speedup with 2 threads against the sequential execution in lines. Although all three have similar performance with block size 128, they behave very different as the block size decreases. For instance, Picos has the worst speedup while Picos++ has the best speedup among all of them with 2 threads. This is due to the asynchronous data communication mode in Picos++. It allows efficient small amount of data exchange by decoupling hardware and software communication. Moreover it allows an easier integration through significantly reducing the amount of interference between the CPUs and the hardware task dependence manager. Therefore, as it can be seen in Table 5.3 and in Figure 5.7, Picos++ always has better speedup than Picos and Nanos++, specially for fine-grained tasks.

5.4.3 Performance and Energy Consumption

5.4.3.1 Synthetic Benchmarks

In this section, the performance of Picos++ is shown in figures 5.8 to 5.10 by using synthetic benchmarks to study the impact of different number of dependences and task granularities on Picos++. Picos++ performance is compared with that of the software-only Nanos++ runtime. All the diagrams in Figure 5.8, 5.9 and 5.10 have two Y-axis, with the primary one (in bars) indicating the execution time in seconds, and the secondary one (in lines) showing the speedup of Picos++ against Nanos++ with 1 or 2 threads.

Figure 5.8 shows results of TestFree and TestChain executing 65536 empty tasks (0 execution time) with different number of dependences. Picos++ and Nanos++ are using the same task creation mechanism. With empty tasks the results highlight the dependence analysis cost in both systems. As the number of dependences (X-axis) increases, we can see that the execution time when using Picos++ (in bars) in both benchmarks barely increases, meanwhile that of the version using Nanos++ increases significantly. This can be also seen as the speedup of Picos++ against Nanos++ with 2 threads (in lines) reaches up to 5.9x and 4x with TestFree and TestChain respectively. TestChain has a tumbling behavior with the software runtime using 2 threads. This effect is due to erratic memory access patterns that are out of the scope of this thesis.

Figure 5.9 shows results with TestFree and TestChain with 65536 tasks, 15 dependences per task and with increasing task sizes. From left to right, the task granularity (X-axis) increases from 30 to 984,000 ns. For tasks around 984,000 ns, the overhead of managing the task dependency graph becomes negligible and both systems deliver nearly

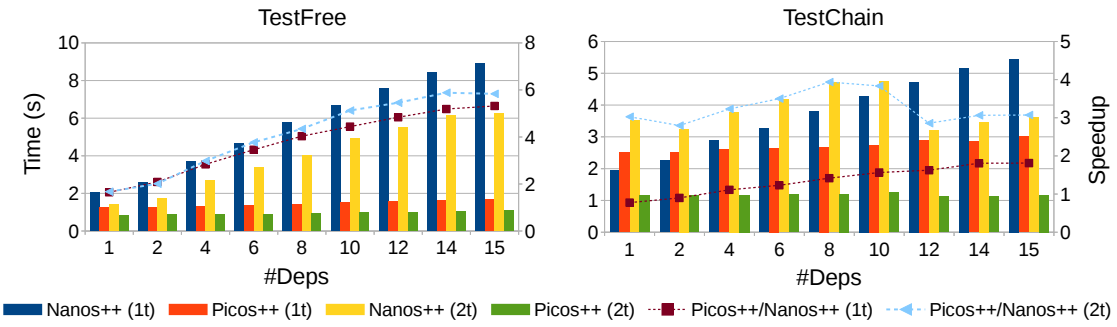


Figure 5.8: Execution time and speedup of TestFree and TestChain with empty tasks

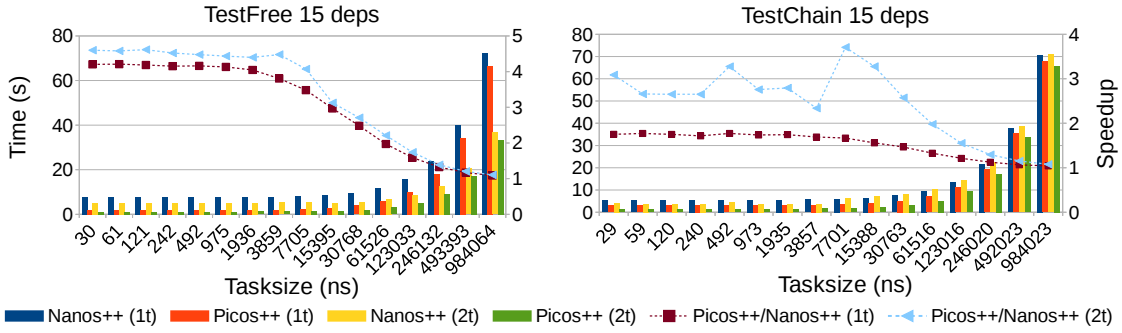


Figure 5.9: Execution time and speedup of TestFree and TestChain with different task sizes

the same performance. However, with tasks of a finer granularity Picos++ has an increasing speedup against Nanos++ that increases with the number of threads. For instance, Picos++ against Nanos++ (Secondary Y-axis, in lines) reaches a speedup up to 4.7x and 3.8x with 2 threads compared to 4.3x and 1.9x with 1 thread. This trend is expected to continue with a larger number of cores.

Figure 5.10 shows results of TestNested with different levels of nested tasks. When comparing with Nanos++, the speedup of Picos++ increases slightly as the nesting level deepens, from up to 1.4x to 1.55x (Secondary Y-axis, in lines) with 4 and 8 levels of nested tasks.

An interesting observation is that with 2 threads Picos++ achieves the highest speedup against Nanos++ with task size smaller than 4 microseconds for independent tasks like TestFree, and around 5 microseconds for dependent tasks like TestChain and TestNested. In addition, Picos++ performs steadily when the number of dependences per task increases, and it achieves better performance with finer task granularity. With more threads, the task size where Picos++ achieves the peak speedup and also outperforms the software-only runtime shall move to the right, thus a larger range of task granularities will benefit

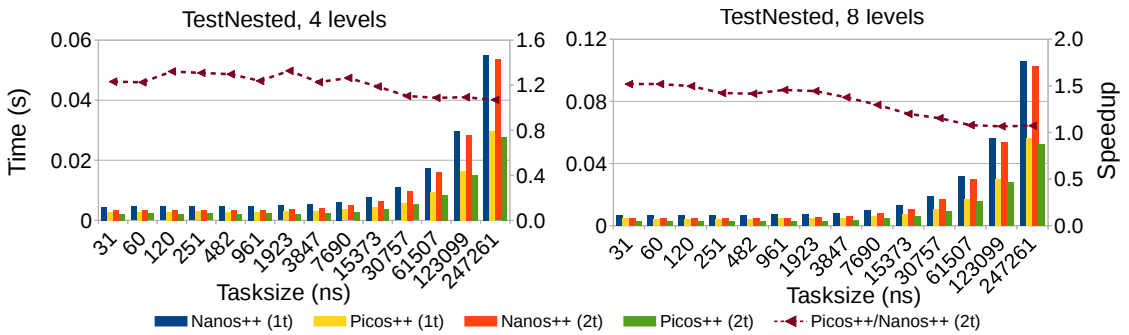


Figure 5.10: Execution time and Speedup of TestNested with 4 and 8 levels of child tasks from Picos++.

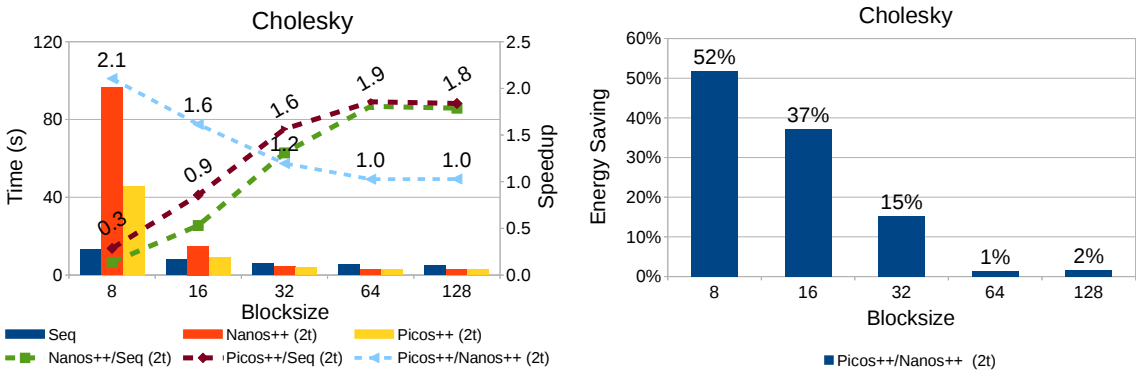
5.4.3.2 Real Benchmarks

In this section, we present scalability studies and energy savings of OmpSs applications Cholesky, H264dec and Multisort with 2 threads. Note that the original implementation of H264dec has two modes of execution with or without nested tasks. The one without nested tasks was used in the original Picos while here the one with nesting is presented.

In Figures 5.11a, 5.12a and 5.13a, each graph has two Y-axis. The primary Y-axis indicates the execution time in seconds (in bars), and the secondary Y-axis shows the speedup (in lines). The X-axis indicates that the task size is increasing from left to right. We also show labeled speedup of Picos++ over the sequential execution, and Picos++ versus Nanos++ runtime with 2 threads in those figures.

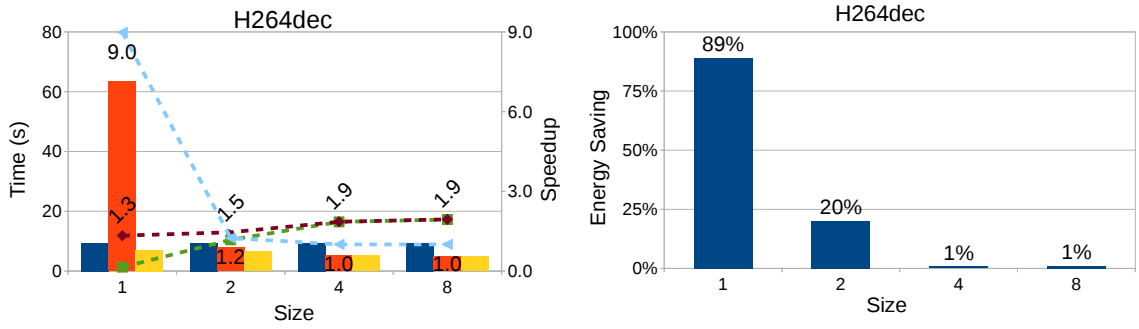
Figures 5.11b, 5.12b and 5.13b show the energy savings obtained when using Picos++ instead of Nanos++ for the corresponding real benchmarks. The legends of the graphs are the same, therefore they are only shown in Cholesky.

For Cholesky, with block size 32, Picos++ achieves 1.6x speedup over the sequential version, and 1.2x over Nanos++ (Secondary Y-axis, in lines). Correspondingly, it saves 15% of energy. With block size 64, Picos++ achieves the highest speedup of 1.9x over the sequential version, similar to Nanos++. With block size 16 and 8, due to the limited system with only 2 threads and the complex pattern of dependences in Cholesky, both Picos++ and Nanos++ degrade, but Picos++ degrades much slower. As it can be seen with block size 8, Picos++ reaches up to 2.1x against Nanos++. The obtained performance results are relevant because those are for only 2 threads where it is easy to exploit all the available parallelism. For bigger systems, Picos++ will show better scalability when the task granularity becomes very fine.



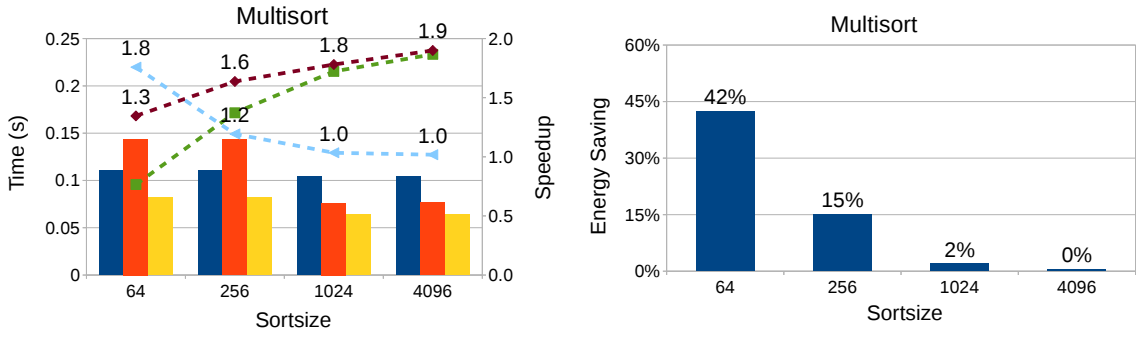
(a) (b)

Figure 5.11: Scalability (a) and energy savings (b) of Cholesky, with 2 threads.



(a) (b)

Figure 5.12: Scalability (a) and energy savings (b) of H264dec, with 2 threads.



(a) (b)

Figure 5.13: Scalability (a) and energy savings (b) of Multisort, with 2 threads.

For H264dec, when compared to the sequential execution, Picos++ has a 2x speedup with size 16 (Secondary Y-axis, in lines). Moreover, with size 2, it is 20% faster than the software-only runtime Nanos++. With size 1, there is a soaring gain for Picos++ against Nanos++ with 2 threads, raising up to 89% of energy savings.

For Multisort, with 1 thread, the execution times of Picos++ are higher than the sequential execution, which is expected because it introduces a lot of communication overheads when no parallelism can be exploited with single thread. However, this degradation in performance using only one thread is significantly overcame by using 2 threads, where Picos++ achieves from 1.3x to 1.9x speedup over the sequential execution, and up to 1.8x over Nanos++ (Secondary Y-axis, in lines). Correspondingly, with sortsize 256, Picos++ saves up to 15% of energy, and with sortsize 64, it saves up to 42% of energy.

5.4.4 Execution Trace Analysis of Multisort

In this section, a 2-threads multisort execution trace using Picos++ is analyzed with Paraver. A similar behavior has been observed for other real benchmarks. Figure 5.14a and Figure 5.14b show different views of the same execution trace. Different colors mean different activities. As explained in the caption of the figures, they show the application task instances and the Picos++ API calls done by the software counterpart runtime during the multisort execution, with 2 threads.

Figure 5.14b has four main Picos++ API calls: `send new task`, `send finished task`, `receive` and `process ready task`. The `process ready task` means checking for ready tasks when there are no ready tasks available. The `receive and process ready task` are displayed with the same color. In the whole trace, the `receive ready task` uses 3.20% and 3.40% of the whole time in thread 1 and thread 2 while the `process ready task` uses 8.93% and 12.79%, respectively.

At the beginning of Figure 5.14a, it can be observed that the main program starts to execute. Meanwhile, thread 2 tries to check for ready tasks before any new tasks are actually created (Figure 5.14b). Afterwards, new tasks are created and thread 1 starts to send new tasks to Picos++, and thread 2 receives some ready tasks. Correspondingly, multisort tasks start to execute. When those tasks end, both threads send finished tasks to Picos++. When the last task finishes its execution, we can see that thread 1 is still checking for ready tasks.

There is a good amount of time when the threads are checking for ready tasks when there are none. For example, at the beginning and end of the trace because there are

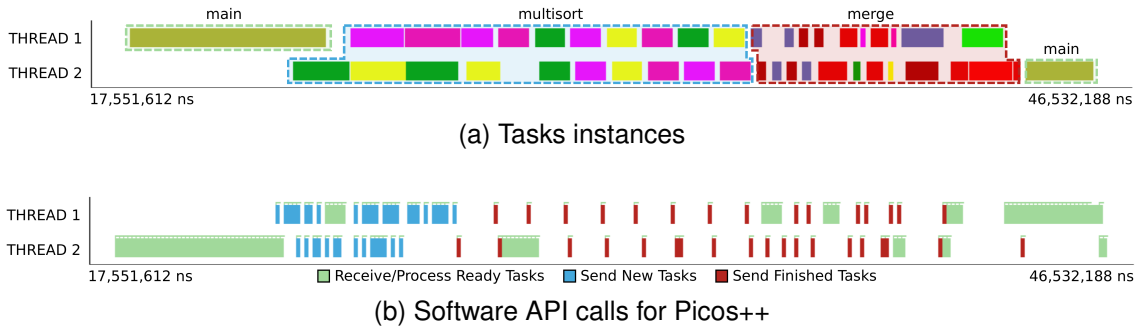


Figure 5.14: Visualization of a real execution of multisort with 2 threads

no ready tasks available at these moments. While in the middle of the execution, the reason for this behavior is that there is no parallelism available. An integrated Picos++ mechanism should be able to generate signals to put the threads to sleep during those times and wake them up later. By doing this, Picos++ could save the energy wasted along these times (21.7% of total time for 2 threads). In this scenario, an integrated hardware manager has the potential to save an additional 10% of energy over Figure 5.13b without affecting the performance. With a large system with more threads, the energy savings can be even more significant.

There is also critical-path tasks information available inside Picos++. Due to the limited system available (with only 2 threads), the current experiments use a simple first-come-first-serve policy for task scheduling. Therefore one interesting future work in Picos++ is to explore the potential performance gain by using critical-path task scheduling with more threads available.

5.4.5 Scalability Discussion

Current implementation of Picos++ shows good results in a fully integrated system with only two available cores. Previous design exploration of the same design, with more threads and no software integration, proved to have great scalability for up to 24 workers for Cholesky (21x speedup) and for SparseLU (24x speedup)[71]. Even more, by using a software cycle-level simulator [87] it has been measured that the same baseline design with one instance of GW, TS and ARB, and four instances of TRSs and DCTs is able to manage up to 256 workers.

To achieve this target, Picos++ should include more logic in terms of more instances of TRS and DCT modules, and communication buffers to distribute and gather tasks and their dependences. On one hand, using more modules linearly increases both the memory

capacity of Picos++ and its internal management flow parallelism. Although the number of modules increases, the characteristics of each module remain the same, therefore the cycle time of the critical path (frequency) should not be significantly affected. On the other hand, increasing the number of components implies adding extra buffers to store all inter-modules messages. This may affect both the operational latency and the frequency depending on the overall implementation size. However, with the current implementation of Picos++, using different lengths of communication buffers has no influence on the performance of real applications, thus the results shown here are with fixed length buffers.

For example, Picos++ with four instances of TRSs and DCTs is four times bigger than the current implementation being able to hold up to 1024 in-flight tasks, and it has the potential to manage 256 workers without introducing overheads in the system with respect to the ideal case [87].

5.5 Summary and Concluding Remarks

This chapter presents Picos++, a general purpose hardware task dependency graph manager for task-based dataflow programming models. The presented design includes for the first time a novel hardware/software co-design that extends the hardware with the capability to manage nested tasks with dependences. In addition, it evaluates real executions of Picos++ as a hardware accelerator support for a task-based programming model, in a Linux embedded system with two ARM Cortex-A9 cores and a FPGA.

Picos++ is a high speed, small and energy efficient runtime accelerator. In a system limited to 2 threads, using Picos++ results on a speedup of more than 1.8x over its software-only counterpart for the most demanding cases of real benchmarks. The corresponding energy savings are greater than 40%.

The gains reported are good for an environment with only 2 threads and they are expected to increase with more resources. Indeed, the hardware accelerated runtime performance speedups are always better than the software-only runtime ones, presenting a better scalability when passing from one to two threads. This effect allows us to conclude that the greater the number of cores the more important the influence of using a hardware task manager would be.

Chapter 6

Heterogeneous task scheduling -Third prototype

The demand for more performance and less energy consumption in computing has lead to a trend towards more heterogeneity in computer architecture. For example, the Cell B. E. [51] consists of a general-purpose core named Power Processor Element and eight accelerator cores named as Synergistic Processor Elements. Another example are GPGPUs [41] which usually are attached to a host processor to be used as accelerators. More examples of this trend are the Imagine [53] and Merriamac [33] stream accelerators, that consist of a grouping of microcontrollers and arithemetic function units. The SARC [63] and the Runnemede [21] heterogeneous architectures which include master and worker cores. Especially significant to our work are the latest Xilinx Zynq Ultrascale+ Devices [86], Intel Stratix 10 SoC chips [48] and the IBM Fabric Power8+CAPI system [44]. When compared with homogeneous processors, heterogeneous ones are able to obtain higher performance with energy efficiency for a large range of applications.

However, they also imply several new challenges. Heterogeneous architectures often have complex hybrid memory hierarchies, which impose serious limitations for the widespread usage of these systems. First, redefining the memory model for heterogeneous systems breaks backwards compatibility, so every new architecture requires adapting scientific and industrial codes. Second, exposing deep and hybrid memory hierarchies to the programmer significantly degrades programmability, as the programmer needs to partition the data, explicitly transfer data between memory spaces, and handle potential data replications. These responsibilities greatly complicate the coding process and, more importantly, require the programmer to have advanced knowledge of the architecture to perform the data management operations efficiently.

One of the most promising solutions to program heterogeneous architectures is using task-based programming models. The task-based extensions were introduced by OmpSs and afterwards added to the OpenMP standard in version 4.5. In task-based programming

models, the programmer exposes the available parallelism of an application by splitting the code in sequential pieces of work, called tasks, and by specifying the data and control dependences between them. With this information the runtime system manages the parallel execution of the workload following a data-flow scheme, scheduling tasks to cores and taking care of synchronization between tasks. In order to manage hybrid memory hierarchies without affecting the programmability of the architecture, in some cases (like OmpSs) the runtime system takes the responsibility of exploiting task annotations to map the data specified in the task dependences to the corresponding memories. So, memory accesses to this data are served more efficiently during the execution of the tasks.

However the cost of using the default software-only runtime system for managing these accelerators is high. Together with the task-dependence analysis, these runtime tasks greatly hinders the potential performance a system can achieve, especially when exploiting fine-grained parallelism.

To accelerate the parallelization and synchronization of workloads in applications and to ease the effort required for data movement in the heterogeneous systems, we further extend the second prototype with a new heterogeneous task scheduling feature. For homogeneous multicores, the software runtime overhead is mainly composed of task creation, dependence analysis and task scheduling. In this environment, task dependence analysis is the most time consuming function [37, 14]. However, for heterogeneous architectures the task scheduling cost is also very high due to the loading imbalance caused by the necessary data movements between different memories and the synchronization between the different heterogenous parts of the system. There are some state-of-the-art works that tackle the load imbalance in heterogeneous systems (in Section 2.2), however their scheduling policies are in software and are themselves often very complex therefore only coarse-grained tasks can be justified to be used to overcome the scheduing overheads. Our third prototype aims to tackle these problems by improving both task dependence analysis and heterogeneous task scheduling in hardware.

In this chapter we present the third prototype of Picos, Picos++ (uses the same name as the second prototype). It is a general purpose hardware for task-dependence management and heterogeneous task scheduling for task-based dataflow programming models. Now with this prototype, we can further improve the performance that can be achieved by the system, and the utilization of the threads that are previously dedicated for task scheduling and data movements in heterogeneous systems. The main contributions of this chapter can be summarized as follows:

- A new heterogeneous task scheduling support in hardware. For each ready task, Picos++ schedules it to a suitable hardware execution unit with the least number of waiting tasks to shorten the total execution time. As a proof-of-concept, different hardware accelerators are developed and integrated with Picos++ to form highly diversified hardware systems.
- A systematic view of the Picos++ system based on a new Xilinx Zynq Ultrascale+ MPSoC platform. It offers insight on how to integrate a hardware task-dependence manager and task scheduler with HW functional accelerators and with host processors when using a parallel task-based programming model.
- Three different studies of scalability and energy consumption when running real benchmarks using the Picos++ system: tasks executing in threads only, HW functional accelerators only and in a mixture of both types. All the evaluation results are compared with a State-of-the-Art software-only runtime.
- Detailed analysis of a Cholesky execution on a system including Picos++, with tasks executed in both threads and HW functional accelerators. This highligths how the application is executed in this heterogeneous system, and future directions to obtain higher energy saving designs.

6.1 Background

OmpSs@FPGA is an effort to support FPGAs as execution units for tasks. It allows programmers to easily create parallel applications which offload some functions to FPGA accelerators. In the source code, the programmer annotates the function that should be offloaded to the FPGA with `#pragma omp target device(fpga, smp) onto(acc id) num_instances(num instances) (copy_deps)`. The source-to-source Mercurium compiler analyzes the whole code and detects two main parts: the host and the FPGA code. The host code is transformed to include calls to the OmpSs runtime Nanos++ to spawn tasks and to transfer the task identification and dependence addresses specified with the `copy_deps` clauses. A specially designed DMA library implements the task and dependences transfers to and from the FPGA. The actual computation data then will be read/write by the accelerators themselves in FPGA after they receive these dependence addresses. Afterwards it is compiled using the GCC compiler. After that, the Mercurium compiler separates the FPGA code, including all High-Level Synthesize

Listing 6.1: Matmul block functions with OmpSs annotation

```
1  #pragma omp target device(fpga) copy_deps onto(0)
2                                  num_instances(4)
3  #pragma omp task inout([bs]C) in([bs]A, [bs]B)
4  void matmulBlock(T (*A)[bs], T (*B)[bs], T (*C)[bs]){
5  unsigned int i, j, k;
6
7  #pragma HLS array_partition variable=A block factor=bs/2 dim=2
8  #pragma HLS array_partition variable=B block factor=bs/2 dim=1
9     for (i = 0; i < bs; i++) {
10       for (j = 0; j < bs; j++) {
11 #pragma HLS pipeline II=1
12          T sum = 0;
13          for (k = 0; k < bs; k++) {
14             sum += A[i][k] * B[k][j];
15          }
16          C[i][j] += sum;
17       }}}
18
19 #pragma omp target device(smp) no_copy_deps
20                                 implements(matmulBlock)
21 #pragma omp task in([bs]A, [bs]B) inout([bs]C)
22 void matmulBlockSmp(T (*A)[bs], T (*B)[bs], T (*C)[bs]){
23    T const alpha = 1.0; T const beta = 1.0;
24    cblas_gemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
25         bs, bsize, bs, alpha, a, bs, b, bs, beta, c, bs);
26 }
```

(HLS) directives regarding communication interfaces and uses autoVivado tool to process it. The autoVivado tool chain, also developed at BSC, it calls the Vivado_HLS and Vivado Xilinx proprietary tools to generate a hardware system to be mapped onto the specific SoC platform on which OmpSs applications can be executed. The `num_instances` directive is used to automatically generate as many accelerators for the annotated function as specified. Afterwards the runtime will take care of using all these available accelerators in parallel when the parallelism of the application allows it.

Listing 6.1 shows an example of OmpSs application Matmul block with two function implementations, one that can be executed in FPGA and the other in SMP. The first function matmulBlock in Line4 indicates that it can be executed in any one of the four instances of HwAccs that will be created at compilation time; the second function matmulBlockSmp in Line22 indicates that it can be executed in SMP and will perform the same function (indicated with implements clause) as the previous one. With this code, if a task is scheduled to FPGA, the first version will be used; otherwise the second version will be executed in SMP. This code with target device fpga is also used to generate the function body of the HW functional accelerator for Matmul as mentioned in Chapter 3.1.3.

Both the software-only runtime and the Picos++ runtime can be used to schedule tasks in the application in Listing 6.1 to SMP and HW functional accelerators.

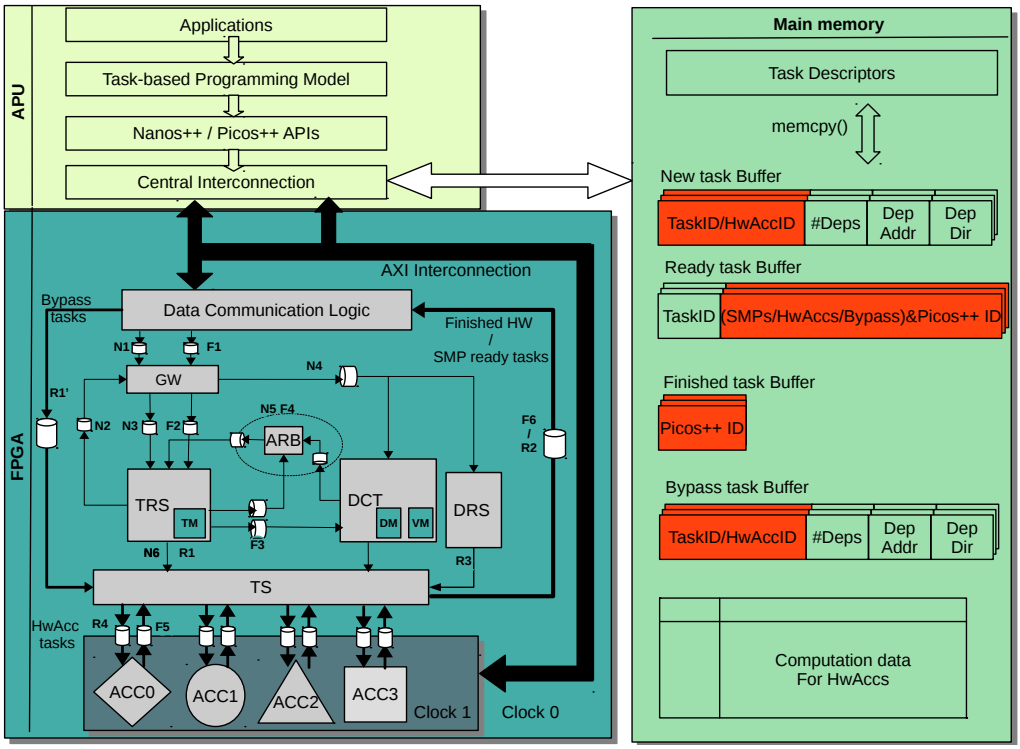


Figure 6.1: Picos++ system organization

6.2 Picos++ System

The third prototype Picos++ adds to the already existing features in the previous prototypes the management of different HW functional accelerators. In order to obtain good performance, Picos++ introduces a new heterogeneous scheduling policy that schedules ready-to-execute tasks to suitable hardware with the least amount of waiting work with consideration of task execution time and the priorities of those accelerators in the system. In this section, we first present an overview of the new Picos++ system. Then we show the hardware support necessary for heterogeneous task scheduling. Finally we show the operational flow of ready tasks in this improved system.

6.2.1 System Organization

Figure 6.1 shows the Picos++ system based on a Xilinx Zynq Ultrascale+ MPSoC. It consists of the Application processing unit (APU) with 4 symmetric ARM cores, the FPGA fabric and the main memory. Applications, programming model runtimes and Picos++ APIs reside in the APU part. Picos++, its data communication, and different HW functional accelerators (HwAccs) are implemented in the FPGA part. There are two clock

domains for the design in FPGA, with the HwAccs in one clock domain and the Picos++ system in the other. All necessary data exchange between these two clock domains are dealt with either asynchronous FIFO queues or clock-crossing (standard method to pass multibits or 1-bit data safely from one clock domain to another).

The APU part includes a central interconnection module that offers a communication interface exposed to the FPGA part. With this system, users can design their own communication network in FPGA and connect through these interfaces to the processors and the main memory. There are two communication channels implemented in the prototype: the first one is used for new, ready, finished and bypass tasks (explained later), where Picos++ and APU share these information by circular FIFOs stored in the main memory. These circular buffers are configurable and can store up to N units representing N tasks. The second communication channel is used to transfer the computation data between the main memory and the internal memories of HwAccs.

New task buffer stores all the information required for Picos++ to manage the inter-task independence. Each unit consists of *Task ID (8 bytes), HWACCID/16bits plus Number of Dependences (in total 4 bytes), and the Dependence Memory Address and Direction of each dependence (12 bytes) for up to 15 dependences*, to add a total of 192 bytes per task. *HWACCID* is used to mark on which hardware devices a particular task can be executed, it can be up to 16 different types.

Ready task buffer, the third prototype extends its functionality into holding more types of messages. Therefore besides holding the ready-to-execute SMP tasks as in the second prototype, it also holds messages that signal the software part of the runtime that a HwAcc has finished executing a normal ready task from Picos++ or a bypass task (this concept is explained later). Each unit includes a *TaskID* field and another field composed of*(SMP/1bit, HwAccs/1bit, Bypass/1bit, reserved zeros/16bits and Picos++ID/14bits)*. Each SMP task message indicated by the SMP bit is used to schedule a task to be executed in SMP as in previous prototypes; Each message indicating the finished execution of a HW task indicated by either the HwAcc or the Bypass bit is used to notify the deletion of its corresponding task/work descriptor in memory to the software part of the runtime.

Finished task buffer stores all the information of finished tasks from threads to update the Picos++ internal task-dependency graph, it is represented by *Picos++ID*. HW tasks that have finished execution in HwAccs are directly sent back through FIFO queues to Picos++ (all in FPGA), thus do not go through this buffer.

Bypass task buffer stores all the information of bypass tasks with the same format as

the new task buffer. The concept of bypass task is introduced to tackle two new situations raised by HW tasks. The first one is a compatibility problem raised by HW tasks that don't use the dependence mechanism (i.e. are synchronously and sequentially executed by the threads). For example, if we want to use the HwAccs without either OmpSs or Picos++, these tasks/workloads need a direct communication mechanism from the APU to the FPGA that bypasses the dependence manager and be directly executed in HwAccs. The second situation appears, for example, in applications with nested tasks like Multisort. As mentioned earlier in nested task support in Chapter 5 section 5.2.2.2, when the first child task has not been read by the hardware because there is a full condition in Picos++, the software support intervenes to avoid a possible deadlock. Afterwards, there are two options for the software to solve the situation, it could either execute the child tasks directly in order by the thread (without allowing it to create more tasks) or submit the blocked task graph as a whole to a software task dependency graph manager. The bypass task channel is added to allow any of these solutions to continue executing HW tasks and avoid a deadlock situation.

Special values of *TaskID* or *Picos++ID* in the new and bypass task units are reserved in order to indicate a valid task, so that Picos++ can read the valid task and process it. Similarly, the 3bits value -*SMP, HwAccs, Bypass* - in the ready task unit are used to indicate three different valid types of tasks that the software runtime can distinguish and process accordingly.

In the Picos++ system, we use a fixed 192 bytes for each new task regardless of its number of dependences, so that the hardware in the FPGA always knows that it has read a complete task and all its dependences. There are also other ways to design the new task communication using variable length of data. For example, using a special value reserved at the end of each variable length as an indication for the end of a new task; or having an additional register to indicate the length of each task. However these ways are much more complicated and costly to be implemented through the AXI interconnection network in hardware, and they are not being proven to be able to improve the performance in our user case. To be worse they are very error-prone for deadlocks due to the delicate synchronization between the software threads and hardware devices.

6.2.2 Data Communication

Data communication is a key factor determining the final system performance. For new, ready, finished and bypass task communication, Picos++ uses the same buffered and asyn-

chronous communication as the second prototype, with the addition of the bypass channel. This communication scheme allows Picos++ to share memory with the threads without support from software side. It decouples the data communication between software and hardware, ensuring a fast and efficient small data exchange.

On the other hand, Picos++ connects to HwAccs through FIFO queues, one ready and one finished queue per HwAcc. Ready tasks are directly dispatched through the ready FIFO queue to the chosen HwAcc. Such ready task include the *TaskID* and all its dependence memory addresses. Each HwAcc reads ready task from the FIFO queue and the input data from these memory addresses, After the computation it writes back the output data to the main memory and the finished HwAcc task is sent back through the finished queue to Picos++. Therefore the ready and finished task exchange between Picos++ and the HwAccs are directly done in hardware, without the main memory as the middleman. This close interaction between Picos++ and HwAccs and their ability to read/write their own computation data result in a very high overall system performance, as we will see in the performance evaluation.

A general description of the system communication of the third Picos++ prototype includes both the software and the hardware side. The software side of the runtime operates as follows: (1) It opens Picos++ for access and allocates four buffers in the main memory. Afterwards, it sends the buffer addresses and lengths to Picos++ and starts the following process. (2) It creates new tasks represented as task descriptors and copies the necessary information to the free spaces in the new task buffer. (3) It checks the ready task buffer. It copies SMP ready tasks to the ready task pool of the worker threads or deletes the corresponding task descriptors in the memory for finished HW tasks. Afterwards it resets the entries in the ready task buffer to an non-valid state. (4) It copies finished SMP tasks to the free space in the finished task buffer for Picos++. Afterwards jumps back to new task processing. (5) When the process finishes, it deallocates buffers and stops Picos++.

In the hardware part, Picos++ has a master module that accesses these buffers by itself. The operational flow of this module follows: (1) Read new task. It reads the *TaskID* field of a unit in the new task buffer. If this is valid, it reads the new task and then invalidates this unit. Otherwise, it tries to write the ready task. (2) Write ready task. It reads the second field of a unit in the ready task buffer. If this unit is non-valid and Picos++ has a ready task, it retrieves the ready task from Picos++ to the ready task buffer. Otherwise it checks for finished task. (3) Read finished task. It reads the *Picos++ID* field of a unit in the finished task buffer. If this is valid, it reads the finished task and afterwards

invalidates this unit. Otherwise if bypass channel is enabled, it checks for bypass tasks. (4) Read bypass task. It reads the *TaskID* field of a unit in the bypass task buffer. If this is valid, it reads the bypass task and then invalidates this unit. Otherwise, it tries to read new task.

6.2.3 Heterogeneous Task Scheduling Support

To support heterogeneous task scheduling in hardware, a new module Dependence Reserve Station (DRS) is introduced to facilitate the read process of dependences for heterogeneous tasks. We also extend the TS module to support task scheduling to up to 16 different hardware units. In addition a new bypass task channel is added to allow tasks to be executed in HwAccs when HW tasks do not use the dependence manager or Picos++ internal memories are full.

6.2.3.1 Dependence Reserve Station (DRS)

DRS includes a internal memory used to save the dependence memory addresses in a simpler memory design to allow easier and faster access for heterogeneous task scheduling. As can be seen in Figure 6.1, DRS receives the same dependence packets annotated with TRS Slots from GW as DCT. For each dependence arrived, the annotated TRS Slot that came together in the packet is directly used as the entry address to the internal memory where the dependence will be saved. When TS wants to schedule a heterogeneous task, all the dependence memory addresses required can be read sequentially from DRS.

6.2.3.2 Task Scheduler (TS)

In this new prototype, in addition to the SMP task scheduling of the TS design in the previous prototype, we added two new functions. Therefore the TS module now is responsible for storing all the ready tasks and schedules them to the suitable hardware execution unit with the least number of waiting tasks; also for receiving all the finished execution tasks from the HW functional accelerators, and organizing them into the ready task buffer for the deletion of task descriptors.

Figure 6.2 shows a simplified scheme of the heterogeneous task scheduling in TS assuming that there are 5 hardware units. It has one ready and one finished task queue for SMP, and one ready and finished task queue for each HwAcc. There are two sources of ready tasks: normal ready tasks that are deemed ready by Picos++, and the bypass tasks.

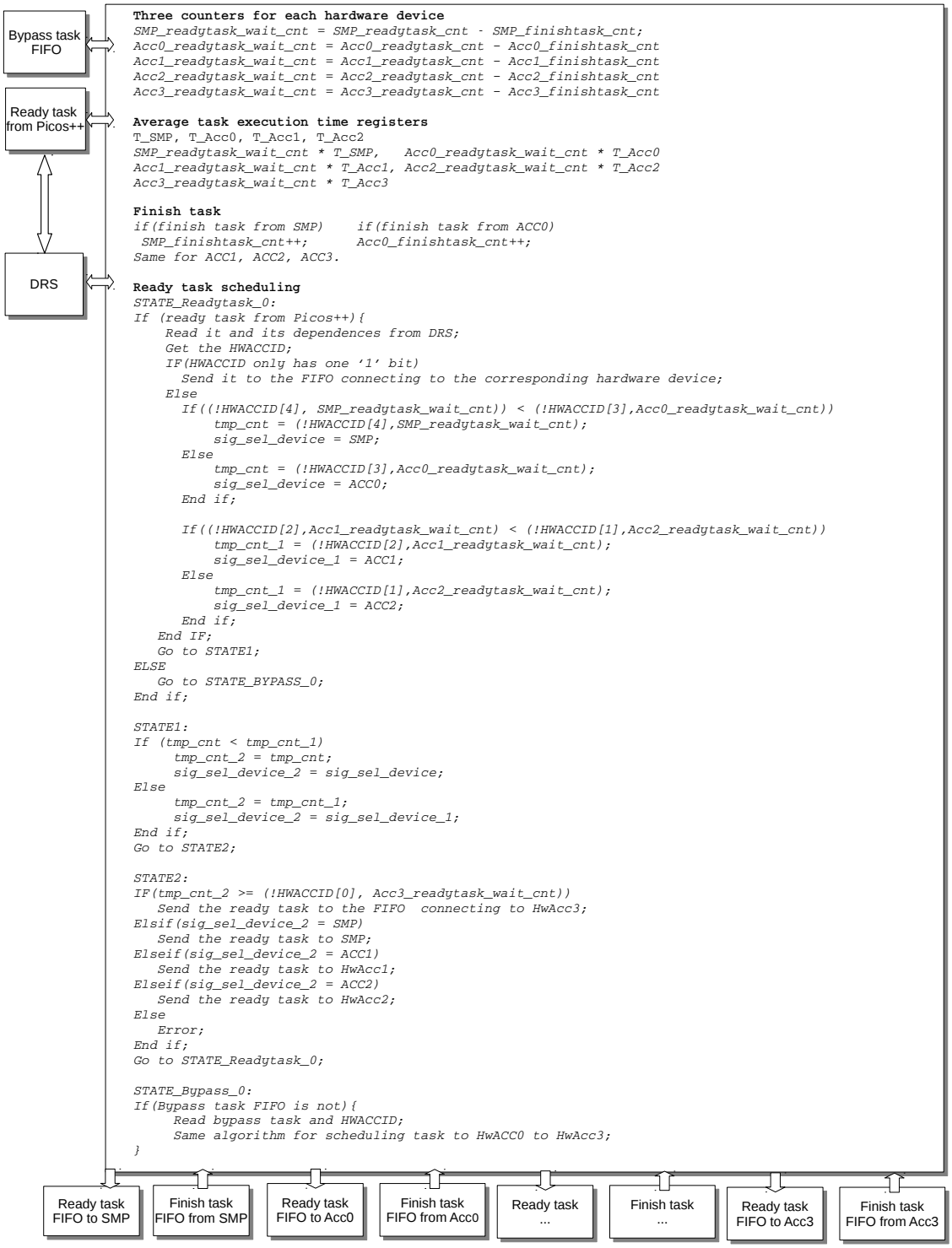


Figure 6.2: Task scheduling to different hardware units

TS checks alternatively for ready task from TRS in Picos++ and from the bypass task queue.

To select a hardware device for task execution, there are four registers associated with each hardware. For example, for HwAcc0, two registers count the number of ready tasks assigned to and the number of finished tasks from this unit, the third register indicates the total amount of work still waiting for this unit; and finally the fourth register counts the average task execution time. When a ready task arrives, Picos first checks its hardware mask to see which hardware devices it can be executed on. If there are several possibilities, Picos compares and selects the one with the least number of waiting work. As can be seen in Figure 6.2, the 5-bits HWACCID correponds to masks for `SMP, HwAcc0, HwAcc1, HwAcc2, HwAcc3`. When the ready task cannot be execute on some of the units, its corresponding `!HWACCID[i]` bits will always be 1 and the `(!HWACCID[i], _readytask_wait_cnt)` will always be the bigger number in the comparison for looking for the smallest amount of waiting work. In the case of multiple identical ones, Picos selects the one with the higher priority. In this prototype, the priorities of different hardware accelerators are fixed and are based on the infrastructure generated. As can be seen in Figure 6.2, the HwAcc3 has the highest priority and the SMP has the lowest priority.

This scheme has a small hardware cost, and it balances the workloads well among different hardware devices considering both different task sizes and the priorities of hardware units in the system. For systems that that have different connections and hardware devices, the priorities can be modified to suit the characteristics of the system, thus achieving a better performance.

6.2.4 Ready Task Operational Flow

In Figure 6.1, besides the New and Finished task processing labels that follow the same pattern as in the previous prototypes, there is also a sequence of Ready task processing (labeled R#) that shows how ready tasks are processed taking into account HW tasks. The process starts with a ready task in the TRS. When TRS has processed all the necessary notifications and marked a task ready (N6), TS reads the message from TRS (R1). If the ready task messages indicates that it is a SMP-only executable task or SMP currently has the least number of waiting tasks, TS schedules it to SMP (R2); otherwise TS reads all its dependences from DRS (R3) and schedules it to one of the HwAccs (R4). For a bypass task (R1'), TS reads all its dependences from DRS (R3) and schedules it to one of the

HwAccs (R4). When a task in the HwAcc finishes, TS reads it (F5) and notifies the HW task finalization through the ready task buffer in the main memory (F6).

6.3 Experimental Setup and Benchmarks

6.3.1 Experimental Setup

This last prototype of Picos++ is coded in VHDL and implemented in a Xilinx Zynq Ultrascale+ MPSoC platform. Its communication logic and all the HwAccs are coded in C with Vivado HLS directives. The final system designs are synthesized with Vivado Design Suite 2016.3.

The hardware platform contains a Zynq Ultrascale+ MPSoC Chip XCZU9EG-FFVC900 [86]. It includes the Application Processing Unit (APU) with 4 ARM Cortex-A53 cores operating at 1.1GHz and a FPGA. Both APU and FPGA are connected to a 4GB DDR4 as main memory. Picos++ and HwAccs reside in FPGA, and can operate at different frequencies ranging from 50 to 300MHz.

The evaluation of Picos++ is done with the task-based parallel programming model OmpSs[35]. OmpSs is supported by the source-to-source Mercurium compiler and the Nanos++ runtime system. Picos++ uses a modified version of Nanos++, as its own software counterpart runtime. Performance tools Extrae and Paraver[26] are also used to analyze the application behavior in the system. Sequential and parallel execution time of OmpSs applications are obtained in the system which operates on Ubuntu Linux 16.04.

6.3.2 Benchmarks

This section shows the synthetic and real benchmarks[25, 79] that are specifically selected to analyze the new features of this prototype of Picos++. Table 6.1 and 6.2 shows the characteristics of the benchmarks and the features of the HwAccs implemented in hardware.

6.3.2.1 Synthetic and Real Benchmarks

A set of synthetic benchmarks TestFree, TestChain and TestNested is selected to test the ability of Picos++. TestFree consists of tasks that are all independent with each other, thus is selected to test the maximum processing capability of Picos++. TestChain is the

opposite with tasks that cannot be executed in parallel at all and thus shows the worst case for Picos++. TestNested is used to test the functionality of nested task support.

Some representative real benchmarks - Matmul, Cholesky and Multisort are selected to focus on testing not only the task dependence management, but also the heterogeneous task scheduling. Table 6.1 shows the number of tasks, sequential execution time and task

Table 6.1: The parameters, execution time and task size of the real benchmarks

Name	Configs	#Tasks	Seq exec time (µs)	Tasksize (µs)
Matmul	(2K, 32)	262144	6820850	26
	(2K, 64)	32768	5463956	167
	(2K, 128)	4096	5435528	1327
Cholesky	(2K, 32)	45760	1232664	27
	(2K, 64)	5984	1087485	182
Multisort	(1M, 256, 256k)	9565	395594	41
	(2M, 512, 512K)	9565	832882	87
	(1M, 1K, 512K)	2397	407648	170

size (granularity) in µs of the applications when executed with different problem size and block size. For example, Matmul with problem size 2kx2k and block size 32x32 has 262144 tasks and takes 6.8 seconds to execute. For Multisort, column Configs indicates the problem, minimal sort and merge sizes. During sequential and parallel execution, the non-recursive tasks of Matmul and Cholesky that are executed in threads are using OpenBlas; for Multisort, the non-recursive basic sort tasks executed in the threads are using qsort.

6.3.2.2 HW Functional Accelerators

Table 6.2 shows the hardware cost (absolute number of units and percentage in FPGA) of the different HwAccs used, and their task size in µs. First row of the table indicates that the task size (amount of time to finish the execution) of the fgemm task using the fgemm32 HwAcc, which is working with 32x32 block size, is 27µs. The same row presents which are the hardware resources that this implementation of the HwAcc needs. The results are obtained with HwAccs at 200MHz. With higher frequency, their hardware cost might change (increase) and their latency will be shorter. For each application, several HwAccs are used at the same time integrated with Picos++ to form a heterogeneous system.

For Matmul, HW functional accelerators with block sizes 32x32, 64x64 and 128x128 were generated: fmatmul32, fmatmul64 and fmatmul128, respectively. For Cholesky, four different kinds of HW functional accelerators were generated to execute the four kernel functions inside the application. They correspond to functions gemm, syrk, trsm

Table 6.2: Characteristics of HwAccs in XCZU9EG-FFVC900

Name	HWACCs				Task size
	BRAM_18Kb	DSP48E	FFs	LUTs	µs
fgemm32	68/3.7%	160/6.4%	19771/3.6%	15559/5.7%	27
fsyrk32	36/2.0%	160/6.4%	19822/3.6%	16149/5.9%	63
ftrsm32	36/2.0%	104/4.1%	11482/2.1%	10875/4.0%	67
fpotrf32	10/0.6%	22/0.9%	3487/0.6%	3302/1.2%	168
fgemm64	74/4.1%	160/6.4%	23887/4.4%	30032/11.0%	126
fsyrk64	42/2.3%	160/6.4%	23849/4.4%	30727/11.2%	270
ftrsm64	42/2.3%	250/9.9%	28734/5.2%	25753/9.4%	314
fpotrf64	28/1.5%	22/0.9%	3514/0.6%	3350/1.2%	981
fmatmul32	68/3.7%	162/6.4%	20106/3.7%	14671/5.4%	27
fmatmul64	138/7.6%	322/12.8%	38770/7.1%	27668/10.9%	105
fmatmul128	287/15.7%	642/25.5%	76147/13.9%	54462/19.9%	497
sort256	68/3.7%	0	20106/3.7%	14671/5.5%	5
sort512	138/7.6%	0	38770/7.1%	27668/10.1%	26
sort1024	159/8.7%	0	47034/8.6%	71124/26.0%	92

and potrf that can be found in the library OpenBlas. In order to study different task granularities, they were generated with block sizes 32x32 and 64x64.

Multisort uses a divide and conquer algorithm, where a multisort task can create 4 nested child multisort tasks when its sort size is larger than a given threshold value. When the sort size reaches the given threshold, it is sorted as a basic sort. In the SMP threads the standard C qsort is used for this basic sort. However qsort is not friendly to be implemented inside FPGA, therefore an even-odd sort is used to generate the HW functional accelerators for this function.

The main purpose of these implemented accelerators shown is not to obtain the fastest execution possible for the applications in FPGA. Instead, they are balanced designs that are quick enough to be considered as fine-grained tasks. Meanwhile their hardware cost are small enough that multiple instances can be fit in the hardware to form a heterogeneous system. They can also be seen as a way of using small HwAccs to solve big problems.

6.3.3 Hardware Resource and Power Consumption

Table 6.3 shows the on-chip resource utilization and the power consumption of Picos++ on the Zynq Ultrascale+ XCZU9EG chip [86]. The resource utilization is obtained from Vivado post-implementation reports while the power consumption is measured during execution time by sampling the electric current in the real platform. All the measurements are done with ARM cores running at 1.1GHz, Picos++ at 100MHz, and HwAccs at 200MHz.

Picos++ and its communication logic together costs around 5% of the available FPGA

Table 6.3: Hardware Resource and Power Consumption

Name	FPGA resource				Power
	BRAM18Kb	DSP48E	FFs	LUTs	in watts
XCZU9EG	1824	2520	548160	274080	
Picos++	87/5.0%	0	5478/1.0%	9793/4.0%	0.07
Comm. Logic	2/0%	0	3421/0.6%	4244/1.0%	0.03
APU					**Watts**
4 ARM Cortex-A53s	-	-	-	-	1.4

resources. This is meaningful as not only it is small enough so it is feasible to integrate it into multicore CPUs, its size also allows us to build a highly heterogeneous platform with Picos++ and several HwAccs in the same FPGA. The 4 ARM Cortex-A53 cores take around 93.3% of the whole chip power. Picos++ and its data communication logic consume a small fraction, around 0.1W.

6.3.4 Power Consumption Measurement

Power consumption is measured through multiplying voltage and sampling current. The voltage is constant at 0.85V for the parts we measured. The sampling current values are obtained through dedicated measurement hardware chips in three main hardware parts: FPGA, APU and the main memory DDR4. As an example, Figure 6.3 shows the power consumption of the APU and FPGA parts in Watts, with an approximate sample of 1/160 µs. The hardware contains APU, Picos++ and one Matmul HwAcc with block size 256x256.

To have an accurate power measurement, different hardware designs were used. For SMP-only tasks, the power measurement of using a software-only runtime is obtained by using a baseline hardware that only includes the APU system. Therefore for software-only, the power consumption of FPGA is considered zero. With Picos++, we consider both APU and only 5% of the whole FPGA (as Picos++ uses 5% of the hardware resources available in the FPGA). For HwAcc tasks, due to the high hardware resource consumption of these HwAccs, we measure the power of the whole FPGA part.

Energy consumption is calculated by integrating the power consumed by APU, FPGA and main memory along the application execution time. Based on these measured energy, the energy savings obtained by using Picos++ versus the software-only runtime or sequential execution are calculated.

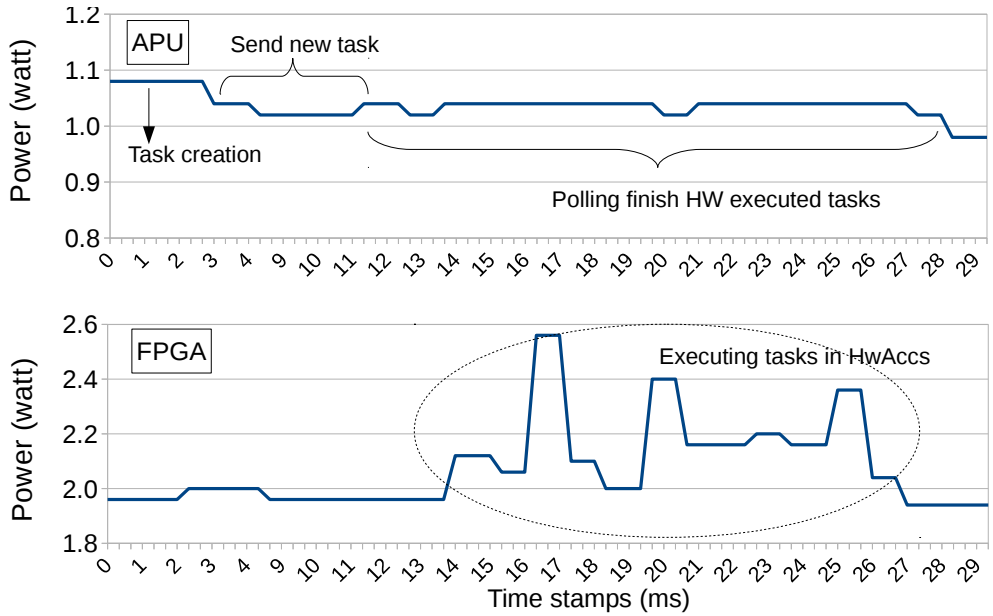


Figure 6.3: Power measurement of APU and FPGA

6.4 Results

This section evaluates the performance and energy consumption of Picos++. Section 6.4.1 analyzes the task and dependence repetition rates in both Picos++ and software-only runtimes. In Section 6.4.2, synthetic benchmarks are executed with different task granularities, dependence patterns and with up to 15 HW functional accelerators to study their performance impact on the Picos++ and software-only runtimes. In Section 6.4.3, real benchmarks are executed with different task granularity and with heterogeneous task management to study the scalability and energy savings of using Picos++. In the same section, more in-depth studies are performed to explore more thoroughly the Picos++ runtime. Finally, visual analysis during application execution is performed.

6.4.1 Task and Dependence Repetition Rate

Table 6.4 shows the task and dependence repetition rates (rrTask and rrDep) of the software-only runtime and Picos++ in 100MHz clock cycles. TestFree and TestChain with 65536 empty tasks (whose execution time is zero) are used to reduce the influence of task execution time, and different number of dependences, from 1 to 15 dependences per task are used to study its performance impact. Row HW-only shows results without any communication latency or software runtime overheads (task creation and scheduling) in Picos++.

Table 6.4: Task, Dependence Repetition Rate in cycles (at 100MHz frequency)

Testcase		**TestFree**		**TestChain**	
Number of dependences		1	15	1	15
HW-only	rrTask	24	243	35	348
	rrDep	24	16	35	23
SW-only 2t	rrTask	1175	5281	1668	3095
	rrDep	1175	352	1668	206
SW-only 4t	rrTask	1055	5272	1706	3118
	rrDep	1055	352	1706	208
Picos++ 2t	rrTask	731	855	1251	1419
	rrDep	731	57	1251	96
Picos++ 4t	rrTask	582	716	1250	1406
	rrDep	582	48	1250	94

Row SW-only 2t and 4t show results obtained by using the software-only runtime with 2 and 4 threads. Similarly Row Picos++ 2t and 4t show results obtained by using Picos++.

As can be seen for HW-only, each task and dependence only takes a few cycles to be processed. In addition, there are two main observations. First, the task repetition rate is proportional to the number of dependences and increases slightly with a much more rigid dependence pattern. For example, for tasks with 1 dependence in TestFree and TestChain, it takes only from 24 to 35 cycles to process a task. Second, the dependence repetition rate is similar for different dependence patterns from 16 cycles to 23 cycles. This is due to the fact that Picos++ pipelines the processing of all the dependences of a task.

The data communication and integration may have a big impact over the whole system performance as can be seen from the larger task and dependence repetition rates. Despite that, in Picos++ system the two observations derived from the HW-only test still hold true. In addition, we can see a clear advantage of managing task dependence analysis in hardware over the software-only approach. For example, with TestFree, Picos++ requires 582 or 716 cycles per task with 1 or 15 dependences, and 48 cycles per dependence. However, with software-only it requires 1055 and 5272 cycles per task. We can also see a slight repetition rate decrease when using Picos++ from 2 to 4 threads. That effect does not shown in the software-only runtime. Next, we move to see some results gathered from real executions by using synthetic applications.

6.4.2 Synthetic Benchmarks

This section uses synthetic benchmarks to examine how the dependences (number and pattern) and the task granularities influence the performance of the Picos++ and the software-only runtime.

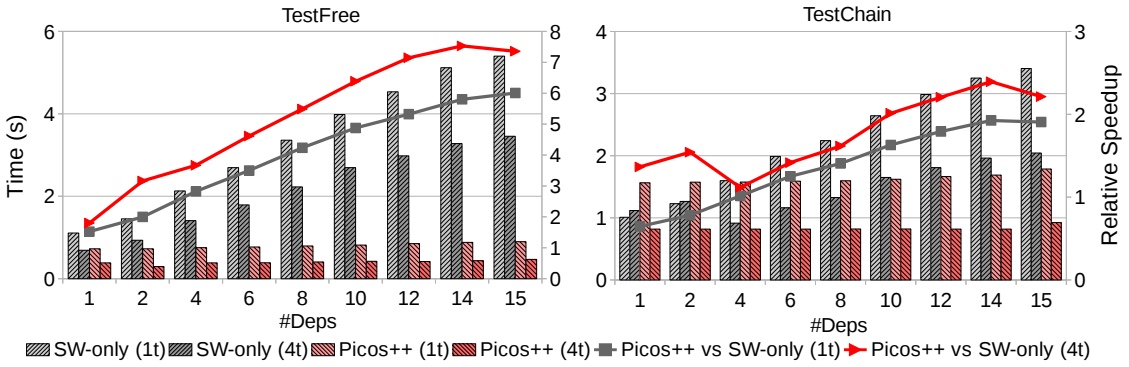


Figure 6.4: Execution time and relative speedup of Picos++ versus the software-only runtime with empty tasks

6.4.2.1 Performance Impact of the Number of Dependences

Figure 6.4 and 6.5 show the results of TestFree and TestChain executions. All the figures presented in this section have two Y-axis, the left one (that applies to the bars) indicates the execution time in seconds, and the right one (that applies to the lines) shows the relative speedup of Picos++ against the software-only runtime with 1 or 4 threads.

Figure 6.4 displays these two benchmarks when executing 65536 empty tasks (tasks with 0 execution time), and with different number of dependences per task. Picos++ and its software-only counterpart are using the same task creation and scheduling mechanisms, and therefore with empty tasks the results highlight the different dependence analysis cost between these two runtimes.

There are two key observations in Figure 6.4. First, Picos++ maintains a nearly equal performance from 1 to 15 dependences per task. Second, it has a much lower dependence analysis cost. With TestFree, Picos++ has a 7.5x relative speedup versus software-only with 4 threads. With TestChain, which opposed to TestFree has no parallelism at all, Picos++ still manages to gain up to 2x versus the software-only approach.

6.4.2.2 Performance Impact of the Task Sizes

Figure 6.5 shows results with TestFree and TestChain with 65536 tasks, 15 dependences per task and with increasing task sizes. From left to right, the task granularities (X-axis) shown ranges from 530ns to 1ms. With 1ms, Picos++ is slightly faster than the software-only runtime. However when the task size decreases, Picos++ outperforms the software-only approach, achieving up to 8x with TestFree and 2.5x with TestChain.

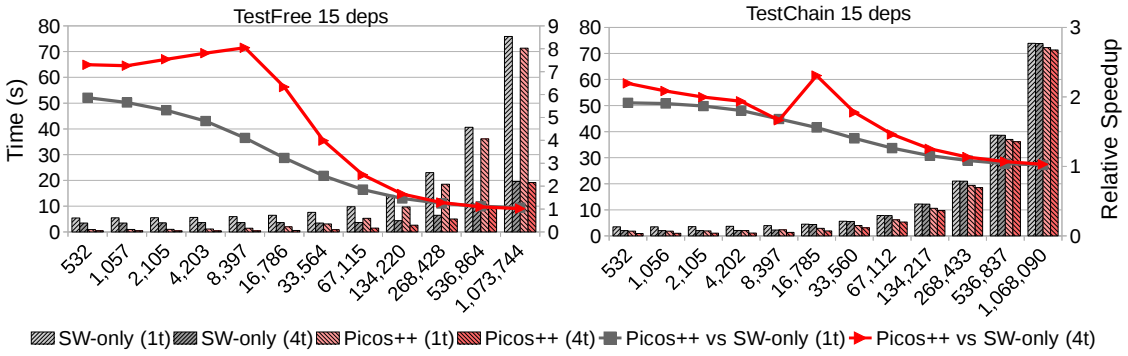


Figure 6.5: Execution time and relative speedup of TestFree, TestChain with different task sizes, with 4 threads

6.4.2.3 Performance Impact of Nested Tasks

Figure 6.6 shows the relative speedup of TestNested with 4 and 16 nested child task per parent task in the nesting level. As can be seen, with smaller task sizes, Picos++ obtains better performance than the software-only runtime, and the same can be said with a bigger number of child tasks per parent.

An interesting point is the task granularity when Picos++ achieves the best performance than the software-only runtime. In Chapter 5, Section 5.4, with the second prototype in a different platform with only 2 threads available, Picos++ achieves the highest speedup peak against the software-only runtime when the task size is around 4K-5K ns with TestFree and TestChain. We predicted that with more threads, due to the faster dependence analysis and reduced thread contention, Picos++ should be able to achieve the highest speedup peak against the software-only runtime from a much larger task size. Indeed, as can be seen in Figure 6.5, with 4 threads Picos++ achieves the best performance against the software-only version starting with task sizes around 8K-16K ns, that is double the task size obtained with 2 threads platform commented above. We expect this behavior to continue with a growing number of cores.

In general, it can be seen that Picos++ has similar performance regardless of the number of dependences per task. In addition, with big task sizes it is as fast as the software-only runtime, and with smaller task sizes it achieves much better performance. Note that these results are obtained by operating Picos++ in FPGA at one-eleventh of the frequency of the software-only runtime in the ARM processors (100MHz to 1.1GHz), with only 4 threads. With more threads or a higher design frequency of Picos++, it can be safely concluded that a much larger range of task granularities would benefit from using Picos++.

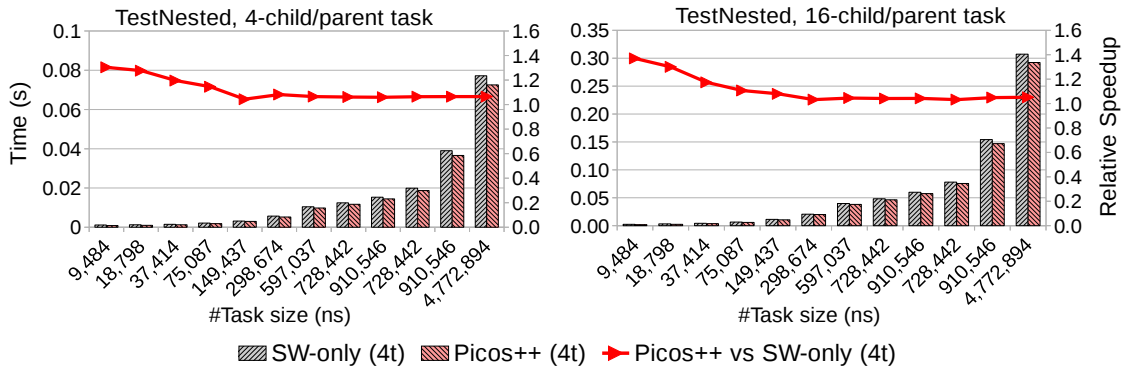


Figure 6.6: TestNested with 4 and 16 child per parent task in each nesting level

6.4.2.4 Scalability with Several Accelerators

Figure 6.7 shows three sets of results for the total execution time (bars, in the left Y-axis) of using the Picos++ and the software-only runtime and the relative speedup comparing these two (lines, in the right Y-axis). The X-axis in each set indicates different task size decreasing from 0.5ms to 160ns. From left to right and top to bottom, the three sets of results are differentiated by Picos++ operating at 50, 100 or 200MHz. All the HwAccs are operating at 200MHz, and all the SMP cores are operating at 1.1GHz. For each set, there are also results for using either sequential or parallel task creation mechanisms (in the cores). In this experiment there are in total 15 HwAccs for task execution, TestFree with 65536 tasks and with 15 dependences per task is executed to obtain all the results.

The sequential TestFree benchmark used to obtain the data in Figure 6.7 is the same that has been used previously in all our experiments. In this version only one thread creates all the tasks that are afterwards executed by the HW accelerators. We have realized that in this particular experiment, as there were so many (15) fast accelerators the element limiting the execution time was in fact the task creation. In order to overcome this problem we have implemented the same program where three of the available threads create the tasks in parallel (using nested tasks). The remaining thread is used to process the finished tasks as this is the fastest configuration to execute this benchmark.

As it can be seen from the total execution time (bars) in Figure 6.7, the software-only runtime benefits from the parallel task creation. However with task sizes smaller than 327,680ns (0.3ms) it is unable to execute the program faster (grey bars). Picos++, on the other hand, when operating at 200MHz, is able to decrease the execution time with task sizes of around 10,240ns (0.01ms) resulting in speedups against the software-only runtime of around 25x. In this case Picos++ is able to fill all the HW accelerators of tasks

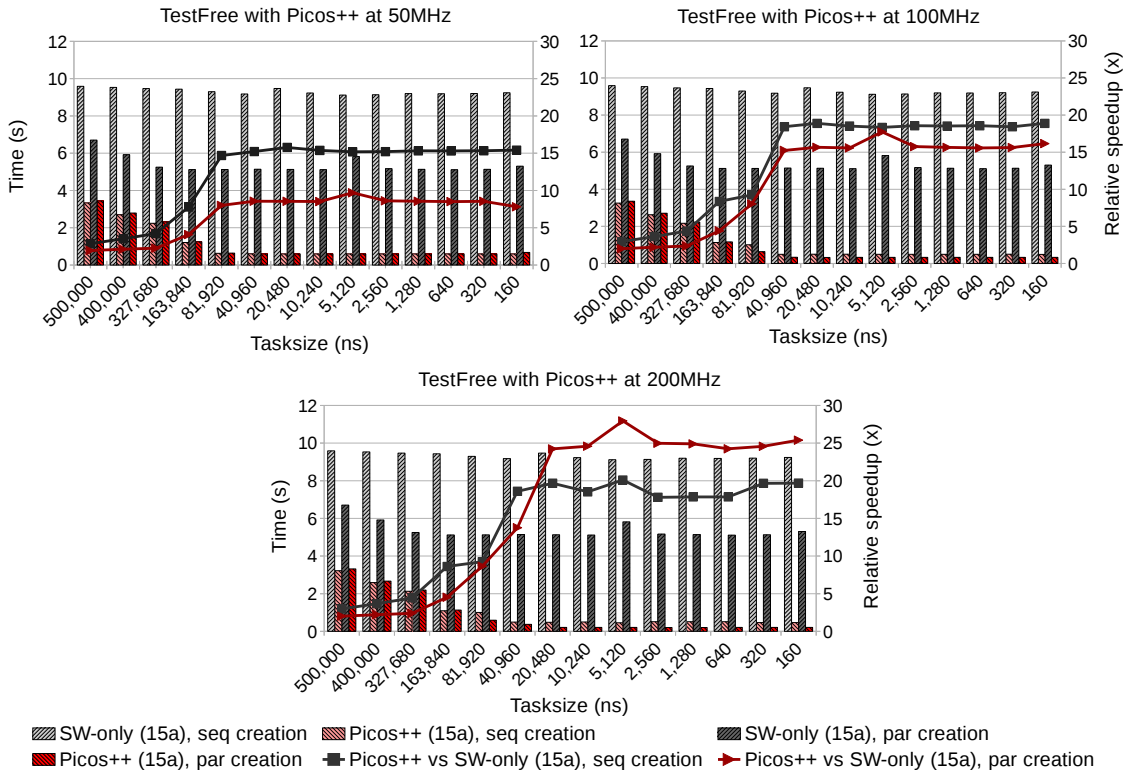


Figure 6.7: Time of Picos++ and Software-only and their relative speedup by using TestFree, with different task sizes and 15 HwAccs

of sizes as short as 0.01ms.

If the results when executing Picos++ at different frequencies are compared (all the other elements in the different plots are exactly the same), it can be seen that the bottleneck in the first plot (Picos++ at 50MHz) is Picos++ as the times with sequential and parallel task creation are the same. With this huge gap between the frequency of the cores (1.1GHz) and Picos++ (50MHz), although it is around 10x faster than the software-only runtime it is not fast enough to serve the parallel task creation with 4 cores at full speed.

If the 100MHz results are observed, it can be seen that Picos++ starts to take advantage of parallel creation. Now Picos++ is exactly two times faster than before and that means that it should be expected to obtain half the execution time for the smaller task sizes. This holds true for the parallel task creation but not for the sequential one. And that means that at 100MHz Picos is able to serve the sequential task creation with 1 core at full speed (the software part is the bottleneck). The time when using the three cores at the same time is larger than one third of the sequential time meaning that in this case Picos++ is still not able to serve the full parallel task creation with 4 cores at full speed.

Finally, looking at the 200MHz results, it can be seen that the sequential time with

Picos++ remains the same reflecting the fact that in this case the task creation is indeed the bottleneck. When the parallel task creation is used, the time is roughly one third of the time with the sequential task creation and larger than half of the time with the parallel task creation with Picos++ at 100MHz. That means that in this last case the software part remains the bottleneck and Picos++ is able to deal with the parallel task creation with 4 cores at full speed. Extrapolating these results, Picos++ operating at the same speed as the cores (at 1.1GHz) should be able to serve around 22 cores creating tasks at full speed with tasks as small as 1861ns executing at the same time.

6.4.3 Real Benchmarks

This section presents the scalability and energy savings obtained using Picos++ with real application executions. The results are analyzed focusing first on the influence of the task granularity and second on the capability of heterogeneous task scheduling.

6.4.3.1 Performance Impact of the Task Granularity

This section shows the impact of different task granularities especially fine-grained on performance and energy. We obtained the results by using both the Picos++ and the software-only runtime, with tasks that are only executed in the ARM cores.

Figure 6.8 shows the speedup, power consumption and energy savings obtained in the analyzed applications when executing them with fine-grained tasks. Each figure shows in the X-axis two sets of problem and block sizes per application (their task sizes are shown in Table 6.1). The Y-axis of Figure 6.8a shows the speedup obtained by the different runtimes when using a different number of physical threads against the sequential execution. In Figure 6.8b the Y-axis shows the average power consumption of both runtimes and sequential execution along the application execution. Finally, Figure 6.8c shows the Energy savings obtained by Picos++ against the software-only runtime and the sequential execution for the same executions.

In Figure 6.8a, Picos++ exceeds the software-only runtime in all the cases. For example, with Matmul with block size 64, Picos++ reaches 3.6x against sequential; in Cholesky with block size 64, it reaches 3.8x and in Multisort with sort size 512, it reaches 3.4x. The performance of block size 32 is worse than 64 in Matmul and Cholesky because there are far more and smaller tasks to manage. These results are relevant due to the limited number of threads and the operating frequency of Picos++. Managing more threads and operating

at a higher frequency, Picos++ should be able to extract much higher performance with block size 32 than with only 4 threads. Despite that, it can also be observed that the gap between Picos++ and the software-only runtime enlarges as the task size becomes smaller from block size 64 to 32. For example, with Matmul with block size 32, Picos++ is 1.5 times faster than the software-only runtime, a bigger difference than with block size 64. A similar gap can be observed in both Cholesky and Multisort.

Figure 6.8b shows the power consumption of the system during Matmul execution. The other applications have similar patterns and therefore only Matmul is shown as a representative. The three main parts of the system: APU, FPGA and DDR are measured during real executions with the board dedicated hardware. As can be seen in Figure 6.8b the power consumption of the APU is proportional to the number of threads that are working, and that of DDR is steady during all the executions. Both the software-only and Picos++ runtime use similar power in these two parts. For FPGA, using Picos++ consumes 0.1 watt more power which explains the small increase in the SUM (SUM=FPGA+APU+DDR) column.

Figure 6.8c shows the energy consumption of all the applications by using Picos++ versus the software-only runtime or sequential executions. When compared to the sequential version, with 4 threads using Picos++ runtime saves more than 60% of energy for all the applications, mainly due to the faster execution. When compared to the software-only runtime, with more threads available and smaller task sizes, Picos++ saves a much higher amount of energy. For example, Picos++ saves 22%, 24% and 42% of the energy compared against the software-only runtime with the smaller task size for the three applications.

Applications with medium size tasks can also benefit from using Picos++. For example, with Multisort with sort size 1K, Picos++(4t) is slightly faster with 3.6x speedup than SW-only(4t) that only reaches a 3.1x against sequential. Correspondingly, with Multisort, Picos++(4t) saves 20% and 65% of the energy consumption compared to SW-only(4t) and sequential execution. To sum up briefly, Picos++ exceeds the software-only runtime in both performance and energy saving. Additionally the gap between them grows larger as the tasks size diminishes.

6.4.3.2 Performance Impact of the Heterogeneous Task Management

This section shows the performance and energy savings of the same applications when tasks are executed in both threads and HwAccs.

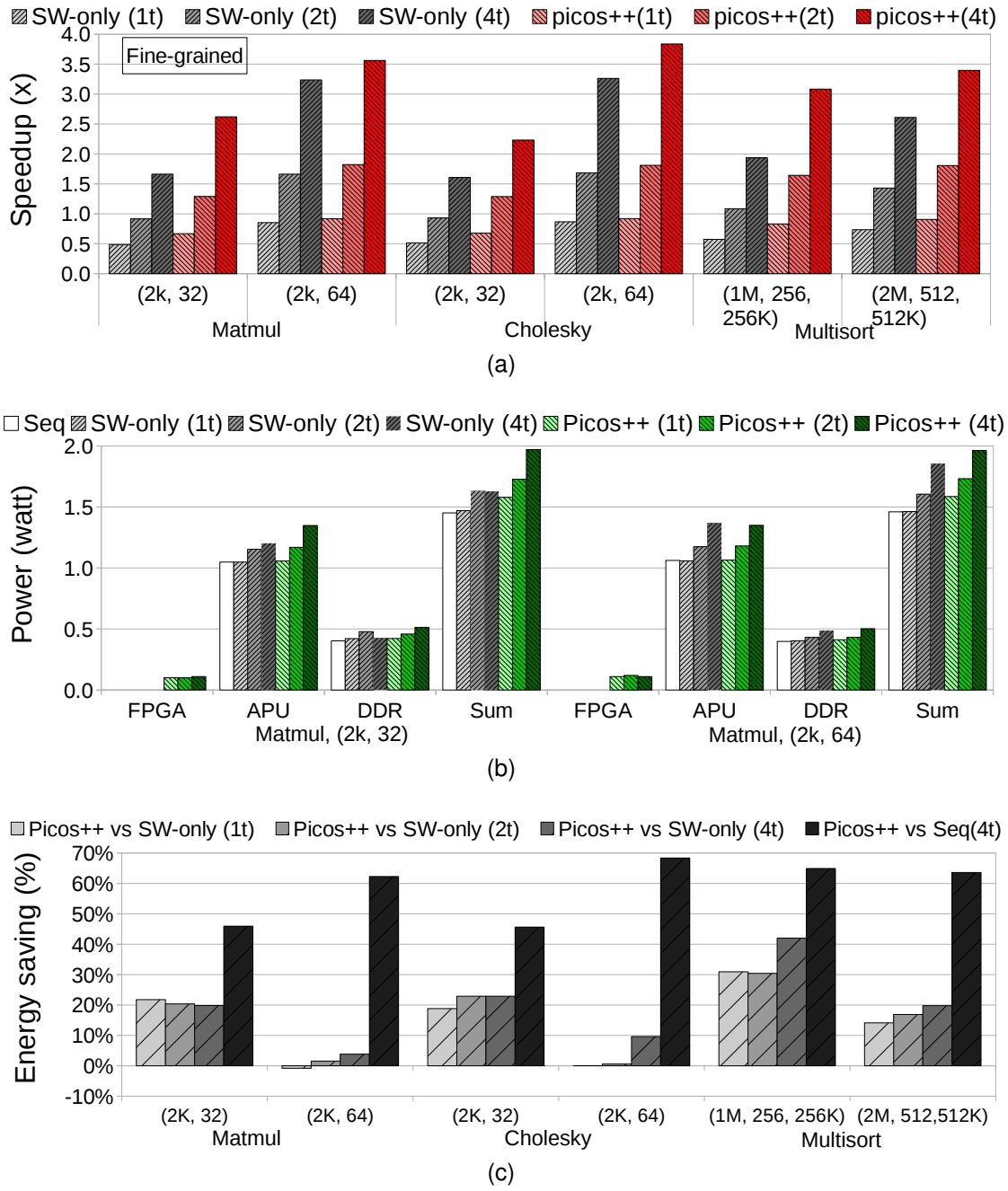


Figure 6.8: Speedup, Power and Energy savings of applications with fine-grained SMP tasks

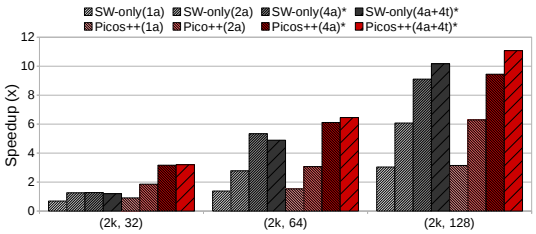


Figure 6.9: Speedup of Matmul, FPGA tasks

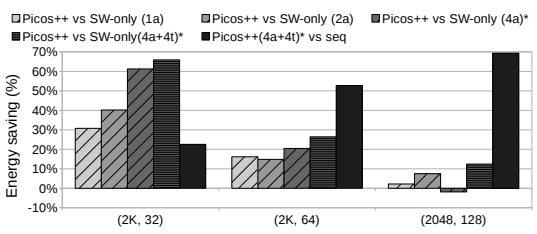


Figure 6.10: Energy savings of Matmul, FPGA tasks

Figures 6.9, 6.11,and 6.13 show the speedup obtained in the three analyzed applications when comparing the parallel runtimes against the sequential execution (in the SMP) and some tasks are executed in hardware accelerators. The legend SW-only(Na) indicates results obtained by using the software-only runtime with tasks that are not executed in threads but in N HwAccs. Legend SW-only(Na+4t) indicates results obtained by using the software-only runtime with tasks that are executed in both 4 threads and N HwAccs. The same logic applies to the legends for Picos++. Special attention should be paid for the legend with "*" symbols. For example, with Matmul block size 128 and Multisort sort size 1K in Figure 6.9 and Figure 6.13, it means that the system contains only 3 instead of 4 HwAccs due to the limited capacity of the FPGA that can not hold 4 of these accelerators. It is also worth to mention that with Cholesky in Figure 6.11, two different sets of HwAccs are used, hence the SW-only(4a+4t) and (4g+4t). This will be explained later.

For all the applications, using a system with HwAccs results in a much higher speedup than using SMP threads only. Just for a brief comparison, in Figure 6.9 with Matmul Picos++(4a+4t) achieves up to 11.2x speedup while when using only SMP threads (in Figure 6.8a) the speedup obtained is only 3.6x. A similar behavior can also be observed in the other two applications. Although the results are good, it is important to state that the hardware accelerators developed in this work were well optimized, but not in the best way. First of all, to continue striving for optimization would have meant a lot of repeated work to obtain all the results again each time we discover a new optimization mechanism, which was not in the best focus spot of this thesis. Secondly, but not least important the accelerators used are perfectly valid as a proof of concept of both the functionality and performance. As the difference in performance would only be better for Picos++ if the accelerators are faster (as this would mean smaller task sizes).

Another conclusion that can be extracted from Figures 6.9, 6.11, and 6.13 is that Picos++ is much better at managing hybrid system than the software-only runtime. With tasks that can be executed in both FPGA and SMP threads, Picos++ achieves up to 11.2x

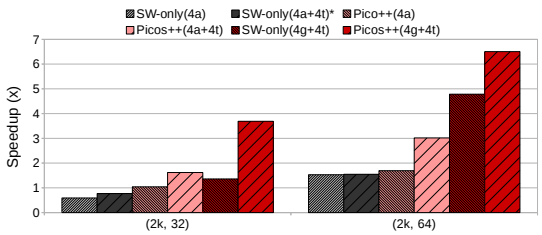


Figure 6.11: Speedup of Cholesky, FPGA tasks

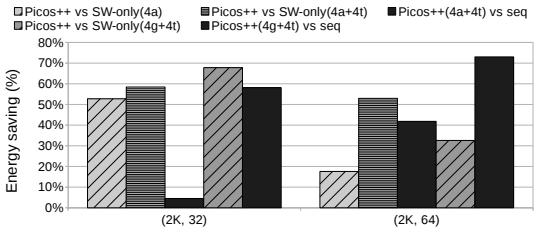


Figure 6.12: Energy savings of Cholesky, FPGA tasks

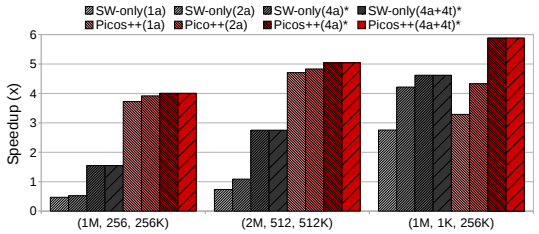


Figure 6.13: Energy savings of Multisort, FPGA tasks

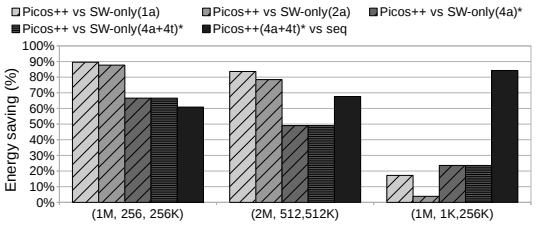


Figure 6.14: Speedup of Multisort, FPGA tasks

speedup against the sequential while the software-only runtime only achieves up to 10.1x. A similar trend can be observed even more dramatically in Cholesky and Multisort. For Choleksy, Picos++(4a+4t) reaches up to 3x speedup while the software-only runtime displays 1.7x. Even better the implementation of Picos++(4g+4t) reaches up to 6.7x while the software-only runtime only achieves 4.8x speedup. For Multisort, Picos++(4a+4t)* achieves nearly 6x while the software-only runtime peaks at 3.6x; in addition, with only 1 HwAcc, Picos++(1a) achieves 3.8x speedup while SW-only(1a) even drops the sequential performance (0.5x).

As in the previous section, the gap between Picos++ and the software-only runtime enlarges as the task size becomes smaller (corresponding task sizes can be found in Table 6.2). For Matmul, the performance gap obtained when going from block size 128 to 32 enlarges from 1.1x to 2.7x faster; for Cholesky, the gap when decreasing from block size 64 to 32 goes from 1.4x to 2.7x and for Multisort, the gap grows from 1.6x to 7.6x.

Correspondingly, Figures 6.10, 6.12 and 6.14 show the energy savings of using Picos++ instead of the software-only runtime or doing a sequential execution for the same applications and configurations.

As can be observed, with a smaller task size and more execution units, Picos++ saves much more energy. For example, for Matmul with block sizes 64 to 32, Picos++ saves from up to 27% to 64% respectively; in addition with Picos++ vs SW-only(1a) to (4a+4t), Picos++ energy saving raises from 31% to 67%. With Multisort, the task sizes of 256

and 512 are very close. As a result, Picos++ saves similar amount of energy for both granularities, up to 90%. For Multisort, it can also be seen that the speedup and energy savings obtained by using 1 to 4 HwAccs do not vary much for the different task sizes. The bottleneck in this case is that most of the execution time is spent on unraveling the nested layers. As a result very few tasks are executed in FPGA and therefore there are simply not enough tasks to feed the accelerators. With Cholesky with block size 32, an energy saving of up to 68% with Picos++ vs SW-only (4g+4t) can be observed.

To summarize, Picos++ is good at managing heterogeneous tasks when taking into account both performance and energy savings. This is partly due to the fast dependence analysis and nested task support as described in the other sections, but also a new factor weights in. The heterogeneous task scheduling in hardware plays an important role in this case as it not only balances the workload among all the devices, but also brings the ready-to-execute tasks much closer to the execution units that reside in FPGA allowing a really fast execution of tasks that depend one on the other.

As mentioned earlier, two different sets of HwAccs (4a and 4g) were used for Cholesky executions. The reason is very application specific, let's consider the three execution cases showed in Table 6.5. The table shows the number of tasks that have been executed in the different implemented hardware units by Picos++: The first case (4a) includes using four different HwAccs only for executing tasks. The four accelerators correspond to four kernel functions of Cholesky application: gemm, syrk, trsm and portf. Therefore each task can only be executed in one of these accelerators. The second case (4a+4t) allows each task to be executed in one specific accelerator or in the threads. Instead, the third case (4g+4t) uses four instances of the same HwAcc (gemm) and uses the threads for the other tasks.

In the first case, gemm tasks executed in HwAcc0 have the highest number among all the four types. In this case, as there is only one accelerator per type of task, there is no parallelism for the same type of tasks during execution. By allowing task to be executed in both HwAccs and threads in the second case, it can be seen that nearly half of the dominate gemm tasks are executed in the SMP threads, which improves 2x the performance when compared to the first case in Figure 6.11. For the third case, by employing four instances of gemm accelerators Picos++ balances better the workloads into the different hardware units, and gains a 4x speedup compared to the first case.

Table 6.5: The number of tasks executed in different hardware in the Picos++ system

Size	Case	#SMP	#HWACC0	#HWACC1	#HWACC2	#HWACC3
(2k, 32)	4a	0	41664	2016	2016	64
	4a+4t	17992	23839	2015	1850	64
	4g+4t	4096	10300	10371	10451	10542
(2K, 64)	4a	0	4963	496	496	32
	4a+4t	2344	2667	493	448	32
	4g+4t	1024	1209	1235	1258	1258

6.4.3.3 Scaling Up the Number of HwAccs

Figure 6.15 shows the speedup of using matmul with problem size 2kx2k and block size 32x32 with up to 12 HwAccs (the maximum number that can be fit inside the available hardware resources). There are two sets of results corresponding to Picos++ working at 100MHz and 200MHz. For each set, there are four bars corresponding to the speedup of Picos++ versus sequential and the speedup of Picos++ versus the software-only runtime, in addition each case is with the sequential task creation used in the previous Matrix multiplication experiments (only one thread creates all the tasks) and the parallel task creation explained in section 6.4.2.4 (three threads create the tasks) are used. The X-axis shows an increasing number of HwAccs used in the system.

In Figure 6.15, when comparing Picos++ against the sequential execution it can be seen that when executing Picos++ at 100MHz, using parallel task creation results in a small boost of the performance of Picos++, allowing it to scale from 6 to 8 HwAccs and obtaining from 3.72x to 5x speedup. With Picos++ at 200MHz, it further scales up to 10 HwAccs and gains from 4.8x to 8.21x speedup. Both the size of the tasks (27 µs in Table 6.2, corresponding to 5440 cycles executed in the hardware accelerators at 200MHz) and the maximum number of accelerators that Picos++ is able to manage at the same time are coherent with the results shown in Figure 6.7.

When Picos++ is compared against the software-only runtime, it can be seen that Picos++ executed at 200MHz and with parallel task creation achieves a 16.2x speedup. The reason of this huge performance gap is that the software-only runtime is in fact slower than the sequential execution by using 1 SMP thread due to the overheads introduced by the taks management: task dependence analysis, task submission contention and heterogeneous task scheduling.

To further explain this effect, Figure 6.16 shows two different traces of the same OmpSs matrix multiplication application (problem size and block size), with parallel creation of the tasks, using the Picos++ at 200MHz (Figure 6.16a) and the software-only run-

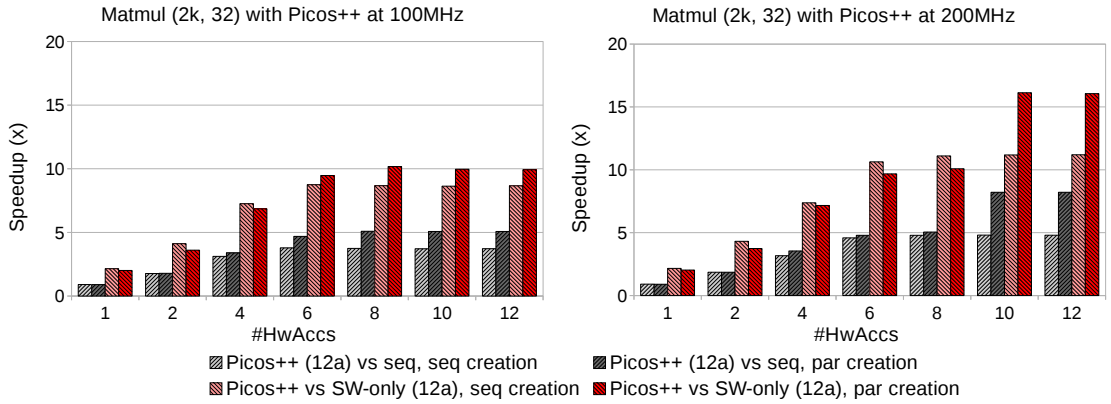


Figure 6.15: Speedup of matmul by using Picos++ and the software-only runtime with up to 12 HwAccs

time (Figure 6.16b). Both trace executions show the thread activities using 12 HwAccs in order to perform the matrix multiplication.

As it can be seen, in both Figure 6.16a and Figure 6.16b, the parallel task creation and the management of FPGA tasks are run among these four threads. The difference is that when using Picos++, it manages the scheduling and finalization of FPGA tasks. In this case the management of FPGA tasks only concerns the deletion of work descriptors in the software part of the runtime. Therefore, the management overhead on those threads is significantly reduced, which results in the performance improvement observed earlier. For the software-only runtime, however, the management of FPGA tasks is much more complex as the threads are required to manage scheduling, finalization and the deletion of FPGA tasks. This overall management, which is expensive and also provokes contention, is translated into a significant increment of the management overhead and the execution time of the application. This explains the performance difference between the Picos++ and the software-only runtime.

6.4.3.4 Potential Energy Savings Analysis

In this section, we show the potential energy savings of Picos++. This is done through analyzing an execution trace of the OmpSs cholesky application using 4 gemm HwAccs. Similar behaviors have been observed for other real applications. Figure 6.17a shows the tasks that each thread is executing at a given moment. As it can be seen, gemm tasks are not shown as they are executed by the HwAcc instead of by the threads. Figure 6.17b shows the Picos++ activities performed by each thread in the same execution. As it can be seen any thread at a given moment is only executing either a SMP task, or a Picos++

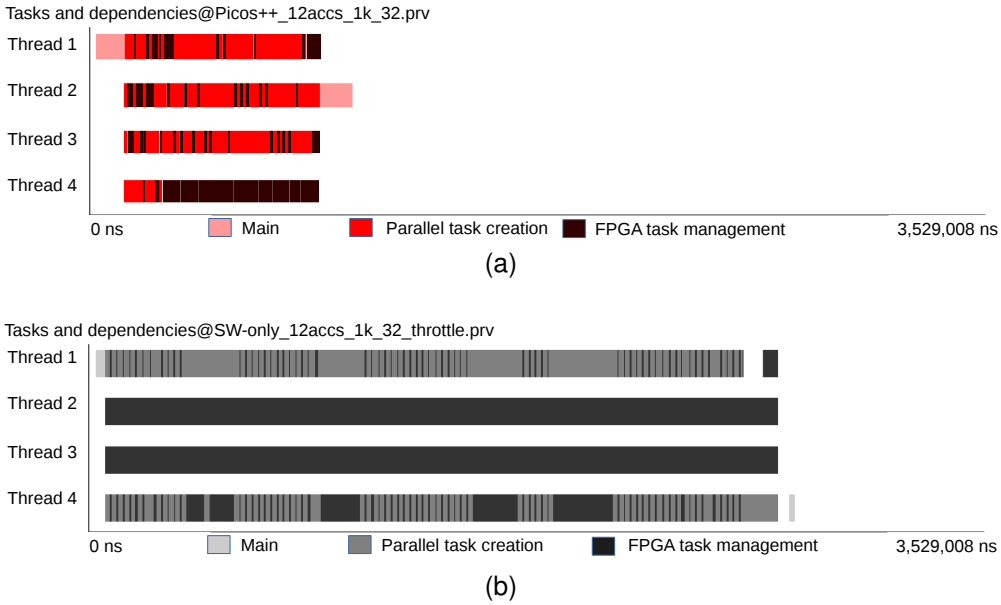


Figure 6.16: Task Instances of Matmul 2k, 32 execution with Picos++ and SW-only with 12 HwAccs

activity or nothing. In addition, we have zoomed in a region in Figure 6.17b to show the series of sequential pollings that the threads perform due to the lack of available tasks. These polling sequences are not continous. All the task dependences are analyzed in hardware and all the omp_gemm tasks are executed in FPGA. This trace shows, for clarity, an execution of a small problem size 256x256 with block size 64x64, which has 20 tasks in total.

Figure 6.17a shows the duration of each task: main, omp_potrf, omp_syrk and omp_trsm. Thread 1 (T.1 horizontal timeline in the figure) is the main thread and launches the `main` function (shown in Figure 6.17a), which creates new tasks (shown in the thread activity in Figure 6.17b), at the beginning of the trace. During this time threads 2, 3 and 4 are actively polling for ready/executed HwAcc tasks. The first SMP task that was deemed ready by Picos++ is omp_potrf, it was scheduled and executed in thread 2 in Figure 6.17a. After omp_potrf, three instances of omp_syrk and then, three instances of omp_trsm are executed in thread 2, 3 and 4. All the omp_gemm tasks are executed in FPGA, so they are not shown in the diagrams timelines. Once a task finishes its execution in a thread, this thread sends a finished task to Picos++. At the end of each task in Figure 6.17a, it can be seen a finished task send action in Figure 6.17b. For all the tasks that are executed in FPGA, there is a executed HwAcc task packet sent from Picos++ to the threads so they can delete the task descriptors in the software part.

In Figure 6.17a, only a portion of time is spent executing useful work. A lot of the

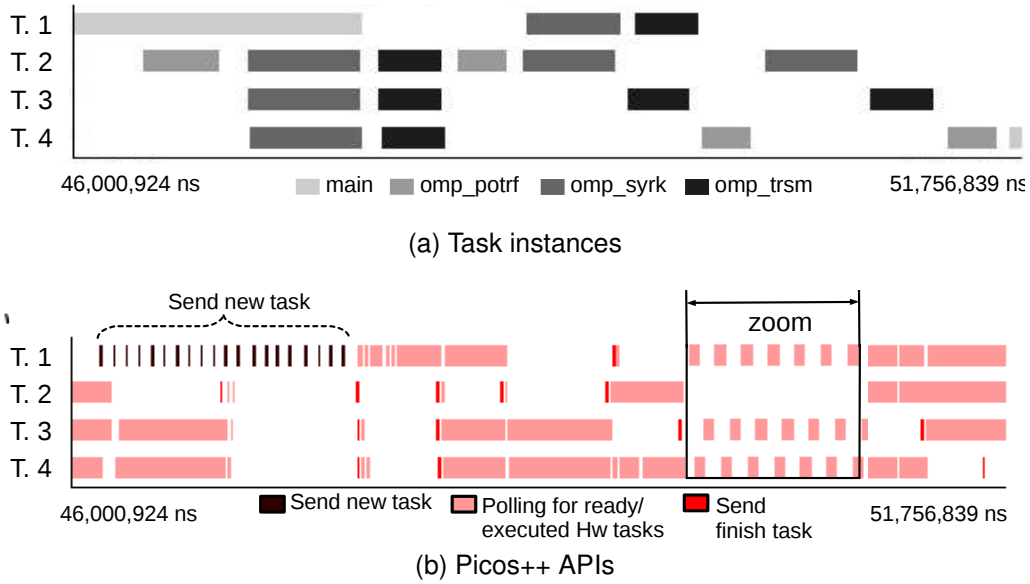


Figure 6.17: Visualization of Cholesky execution (With Problem size 256, block size 64, with Picos++(4g+4t))

time is wasted on idle polling for ready/executed HwAcc tasks as shown in Figure 6.17b. To quantify the execution time where the threads are actually doing useful work of the application or runtime, it has been measured in the trace. Table 6.6 summarizes the percentage of time of all the useful functions in the threads and in the HwAccs for a real problem with a bigger problem size 2Kx2K.

As it can be seen in the *Tasks* category, the main task is executed mainly in Thread 1 and consumes 97% of its time. The other threads execute mainly the potrf, syrk and trsm functions when they are available. Gemm tasks are only executed in HwAccs (HwAcc0 to HwAcc3) and do not consume any execution time in the threads but use 13.34%, 56.52%,

Table 6.6: Useful time consumption with problem size 2048, block size 64

Category	Name	T1/HwAcc0*	T2/HwAcc1*	T3/HwAcc2*	T4/HwAcc3*
Measured Useful Time in the trace					
Tasks	main	97.00%	1.24%	0.00%	0.00%
	potrf	0.14%	0.78%	0.76%	0.75%
	syrk	0.28%	15.32%	14.47%	15.00%
	trsm	0.26%	20.79%	20.79%	22.64%
	gemm*	13.34%	56.52%	56.52%	60.89%
Picos++APIs	New task	33.00%	0.00	0.00	0.00
	Polling ready	0.81%	19.19%	19.16%	18.49%
	Successful ready	0.05%	5.89%	5.52%	4.78%
	Finished task	0.04%	3.08%	3.07%	3.06%
Potential energy savings time					
HwAccs	omp_gemm*	86.66%	43.48%	43.48%	39.11%
Threads	Upper bound	0%	48.63%	54.75%	53.11%

56.52% and 60.89% of the execution time in each accelerator respectively.

Category *Picos++ APIs* shows the time that each thread expends dealing with Picos++ activities. `New task` includes the time of copying necessary data from newly created Task Descriptor into the new task buffer. `Finished task` includes the time it takes to copy the Task Descriptor of the tasks that finished executing in threads into the finished task buffer; `Polling ready` includes all the time that threads spent for busy checking for ready or finished executed FPGA tasks. As it can be seen, `Polling ready` is the dominant factor, due to the fact that threads have no knowledge of when there is a ready task and basically they are busy checking when there is no available parallelism. `Successful ready` shows how much time consumed for the actual successful read ready task action. New, Ready and Finished task buffer are described in Section 6.2.1.

Finally, in section *Potential energy savings time* in Table 6.6 it is summarized how much of the traced execution time the Hardware functional accelerators or the Threads are doing nothing. Picos++ has knowledge about when there is a ready task that can be executed in FPGA or threads. Therefore it could issue commands to turn off the HwAccs and to put threads to sleep. When there is work to be done, it can generate signals to active FPGA and wake up the threads. There exist well documented methods such as clock gating for turning on/off FPGA functions rapidly, and commands for sleeping/waking commercial processors. Theoretically in this execution the HwAccs could be turned off for more than 40% of time. The maximum time that can be saved in each thread is computed by removing of the total execution time all but the useful functions including task creation, execution and Picos++ activities. Taking into account this upper-bound, threads 2 to 4 could be put in low power mode for approximately half of the execution time.

6.4.3.5 Picos++ and HPC

In this section, the performance results obtained when using the fastest hardware functional accelerators developed at the moment are shown. Although the accelerators used could surely be improved by a low-level programmer (a work that it is out of the scope of this thesis), the goal is to show how the use of a hardware task manager improves the performance obtained by a heterogeneous system even for the best known case of the software only approach.

Figure 6.18 shows the Gflops (Y-axis) obtained by both the software-only and Picos++ runtime when computing Matmul with 3 HwAccs of block size 128 each. From left to

right, and for both the software-only and Picos++ runtime, the bars represent the performance result (in Gflops) when using only FPGA task acceleration, FPGA acceleration and 1 SMP thread to execute SMP tasks, FPGA acceleration and 2 SMP threads, FPGA acceleration and 3 SMP threads, and FPGA acceleration and 4 SMP threads executing SMP tasks. Matrix Multiplication tasks in the SMP uses the optimized version of the OpenBlas sgemm (single precision matrix multiply) implementation. The threads are operating at 1.1GHz, Picos++ at 100MHz and HwAccs at 300MHz. It is important to state that this configuraction (3 HwAccs of block size 128) is the one that obtains the best performance for this problem in this system when using the software-only runtime.

As can be observed in Figure 6.18, the software-only runtime requires 2 threads to manage the task scheduling for 3 HwAccs, which causes a drop of performance when 3 or 4 SMP threads are enabled for task execution due to oversubscription (the same thread is used both to compute tasks and send tasks to the hardware accelerators). On the other hand, Picos++ schedules tasks to HwAccs without help from the threads, which greatly improves the overall performance. There is a small slowdown when using HwAccs with 1 additional SMP thread due to load unbalance in the scheduling policy as task execution time in HwAcc is smaller than in the threads. However, with more SMP threads in the system, this unbalance disappears. This unbalance with one thread could also be solved by enabling the average task execution time policy together with the smaller number of waiting tasks in Picos++ scheduling as mentioned in earlier sections.

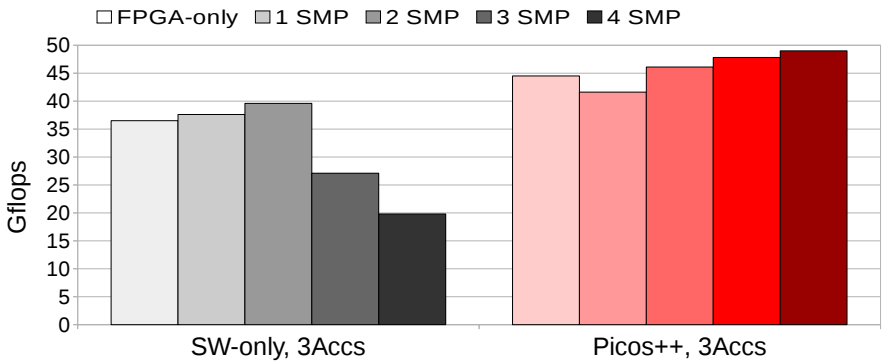


Figure 6.18: Gflops of Picos++ versus the software-only runtime with 3 HwAccs@(300 MHz) with Matmul with problem size 2k, block size 128

As it can be seen, Picos++ obtains more Gflops when more hardware resources exist in the system and better performance than the software-only runtime. For example, the software-only runtime with 3 HwAccs achieves 39.6 Gflops while the Picos runtime

obtains up to 49 Gflops. The whole system executing this problem has a power consumption of only 5.74 watts, which results in 8.54 Gflops per watt. To show what this value means, the same application was run on different machines. A Intel(R) Core(TM) i5-3470 which has up to 4 threads operating at 3.20GHz achieves 0.51 Gflops per Watt. A Intel(R) Xeon(R) CPU E5-2020 V2 which has up to 24 threads operating at 2.1GHz achieves 4.14 Gflops per watt and a Intel(R) Core(TM) i7-4600U CPU which has up to 4 threads operating at 2.1GHz achieves 4.75 Gflops per watt. All of the Intel machines were programmed using MKL sgemm to perform the matrix multiplication.

6.5 Summary and Concluding Remarks

This chapter presents the new features of Picos++, a hardware accelerated runtime for task dependence management and heterogeneous task scheduling, for task-based programming models. It includes all the features presented in the previous prototypes plus the ability to schedule heterogeneous tasks and accelerate the synchronization with the HwAcc tasks. This new capability helps to further free threads for other useful functions and improves the performance over heterogeneous platforms.

In addition, the current implementation of the Picos++ system is built on one of the Zynq Ultrascale+ MPSoC platform. Picos++ is connected with the SMP and numerous HW functional accelerators. It is also integrated with the programming model OmpSs and Ubuntu Linux 16.04.

The Picos++ runtime exceeds the default software-only runtime in terms of both speed and energy efficiency. The smaller the task size or the more hardware resources available in the system, the higher the performance that Picos++ is able to achieve compared against the software-only runtime.

In a system with 4 threads, with tasks that can only be executed in threads, using Picos++ results on a speedup close to 4x versus sequential execution with real applications; more importantly, it gains up to 2x over its software-only counterpart. When dealing with tasks that can be executed in both threads and HW functional accelerators, with up to 4 threads and 4 such accelerators in the system, Picos++ achieves up to 11.2x speedup versus sequential version; and when compared to the software-only runtime, it achieves up to 7.6x speedup with real benchmarks and could saves up to 90% of energy with fine-grained tasks. With even more hardware execution units, with up to 12 accelerators, Picos++ achieves up to 16.2x when compared with the software-only runtime system.

The fast dependence analysis and task scheduling in hardware, in addition to the reduced thread contention, are the key factors that contribute to the high performance and energy savings of this design. Due to the experiments performed with a higher number of accelerators, it is expected that in larger systems with more threads and hardware execution units the performance gains and energy savings obtained against current software-only runtimes will be even more significant; making Picos++ a very suitable piece in future many-core systems.

Chapter 7
Conclusions

This chapter summarizes the main conclusions and contributions of this thesis and presents the future research lines opened by this work. Afterwards it acknowledges the financial support.

7.1 Goals, Contributions and Main Conclusions

The end of Moore's law has significantly shifted the aims and landscape of processor designs. In the early 2000s, to overcome the stagnation of single-core processor performance, processor design entered a new era with multicore and many-core processors. Shared memory multiprocessors are one of the most representative types of chip multiprocessors with examples such as the IBM POWER4, Intel Core Duo, AMD Opteron, or more recently Intel Xeon Phi and IBM Blue Gene/Q. Such architectures can provide the desired performance gains by exploiting high Thread Level Parallelism or Task Level Parallelism from parallel programs. However, they also introduce significant challenges for adapting from sequential to parallel computing. Parallel computing requires a huge mount of programming effort to detect, distribute and synchronize parallel workloads/-tasks among the available threads/resources.

Moreover, current processor designs exhibit a trend towards heterogeneity for even higher performance and energy efficiency. Architectures as Cell B.E., the SARC architecture, Big-little cores, GPGPU, Xilinx Ultrascale+ MPSoC series and Intel Stratix-10 SoC series are all examples of this trend. While these architectures are able to offer potentially higher performance at a low energy cost than traditional multicore and many-cores, they also raise new complicated problems. Heterogeneous systems often are augmented with complex memory systems, such as a mix of global coherent and local non-coherent memories. Once more, programmers have to be aware on the underlying hardware ar-

chitecture in order to obtain real benefits. They are even often responsible for explicitly transfer data between those memories and handle potential data replications.

To tackle those challenges, to automate the parallelization process of workloads/tasks in applications and ease the effort required for managing multicore, many-cores and heterogeneous systems, task-based programming models have appeared with new proposed solutions. Significant examples of such programming models are OpenMP, Intel's TBB, OmpSs, Codelets, StarPU, Charm++. They are simple to use and are powerful enough to gain high performance by exploiting task parallelism and heterogeneity using coarse or medium grained tasks.

However, there are no trivial solutions to exploit parallelism with fine-grained tasks. On one hand, fine-grained tasks are favorable for exposing higher parallelism within the same application, and additionally they make it easier to manage load balancing among different hardware execution units. On the other hand, task-based programming models like OpenMP and OmpSs usually employ software-only runtimes as their default ones, which have large overheads and thread contention, leading to performance degradation. The critical overheads are mainly task-dependence management and heterogeneous task scheduling.

To overcome these disadvantages, we propose a hardware architecture Picos to accelerate these two critical runtime functions. In Picos, a hardware task-dependence manager manages all the task dependences with different patterns. A hardware task scheduler schedules tasks to a hardware device (including SMP, FPGA, ASIC and GPUs) with the least amount of active waiting work. It also takes care of the load balancing and task synchronization among SMP and HwAccs. Three prototypes of our proposal have been developed and studied. They are coded in VHDL and implemented in real commodity hardware platforms.

The first prototype is a hardware task dependence manager, which has been implemented in a Xilinx Zynq 7000 series SoC. It is connected to a 2-core ARM Cortex A9 processor, with bare-metal OS integration. With 24 simulated workers, and running real task-dependence analysis in Picos, it scales up to 21x speedup with 24 workers with real benchmarks. It costs only a tiny fraction of hardware resources and of the whole power consumption.

The second prototype Picos++ extended Picos with an exciting new feature for nested task support in hardware. To the best of our knowledge, this is the first time that such a feature has been support fully in hardware task dependence managers. In addition,

a different communication scheme is employed to allow a faster data communication between Picos++ and ARM processors. The second prototype is fully integrated in not only hardware, but also with a State-of-the-Art parallel programming model, and with a Linux OS. Therefore, real applications are directly executed in the Picos++ system to obtain performance and energy results. Since it has been evaluated with real executions in real hardware, it is limited to only 2 worker threads. However the results are promising. With only 2 threads available, it gained more than 1.8x over its software-only counterpart, and saves more than 40% of energy.

The third prototype, with the same name, includes a hardware task dependence manager and a heterogeneous task scheduler. The heterogeneous task scheduler receives ready tasks from the task-dependence manager and then decides to schedule them to hardware execution units that have the estimated earliest finish time. The third prototype is implemented in a Xilinx Zynq Ultrascale+ MPSoC chip. In a system with 4 SMP threads and up to 4 different types of HW functional accelerators (task execution units), it achieves up to 7.6x speedup for real benchmarks, and saves up to 90% of energy comparing to the software-only runtime. With up to 12 HW functional accelerators, it even achieves up to 16.2x speedup over the software-only alternative.

Generally speaking, this work aims to be a proof-of-concept of the suitability of hardware runtime for current processors. The results obtained not only demonstrate that such runtimes are possible, but also exhibit unexpectedly large performance gains for very small systems. Even more, even in systems with a small number of cores they showed a growing trend in both performance gains and energy savings. This trend is expected to continue and even increase along with system complexity. As so, we suggest that, as many other solutions in computer architecture before, runtimes should start moving part of their complexity to the hardware to further separate the way a program is parallelized from the underlying hardware and to avoid becoming the next bottleneck in computation.

7.2 Future Work

Along with the further development and analysis of the designed prototypes, there are several future directions that we would like to investigate in the future.

- **Fast sleep/wake-up in hardware**. During the execution analysis of real applications on the second and third prototype, there is a recurring pattern that the threads and other hardware execution units were not given enough information about the

parallelism in the application. They are kept on working in a busy-waiting loop when they could be put into a low energy mode without influencing the performance. Since Picos++ has all the task-dependence information inside, we believe that it can offer a very fast sleep/wake-up mechanism in hardware, allowing to build a highly energy efficient system.

- **Towards more heterogeneity**. Our proposals in this thesis are real implementations and integrations, thus are limited by the hardware platforms that we have. Currently the Picos++ design is integrated with SMP and HW Functional accelerators in FPGA. However the design can be adapted to manage other devices such as different types of processors, ASICs, GPUs, and other accelerators. From the performance and energy saving results we have obtained so far, we believe that Picos++ can further boost the performance of other hardware architectures for parallel applications. Additionally since the Picos++ design costs only a tiny fraction of hardware and power, it is feasible to integrate it with architectures that target different goals (and not only high-performance) without much area and power overheads and with significant gains.

- **Towards many-cores**. In a previous work, a design space exploration estimated that the current size of our prototypes is able to manage up to 8 cores without performance losses over the ideal case. A larger design with four times more modules (simulated by using a C model) was estimated to be able to manage up to 256 cores without degrading performance. The current prototypes have proved to be able to scale up to more than 8 cores in real situations, therefore it would be really interesting to see the impact of a larger size design in more complex systems.

- **Towards larger systems**. It would be really interesting to develop Picos++ to manage systems composed of more than one node. Current programming models runtimes are being extended to manage multi-node systems by integrating complex memory systems such as PGAS (Partitioned Global Address Space). We believe that significant savings can be obtained in the synchronization time by integrating the data communication between nodes (i.e. the network) with Picos++.

- **Other task-based programming models or other usage**. Currently Picos++ is integrated with the OmpSs programming model. However, since It only requires general information such as task identification and dependences, it can be integrated

with other task-based programming models such as OpenMP, codelet, StarPU, etc. Moreover, as long as a piece of work can be expressed as small workloads with dependence relations between them, theoretically Picos++ can manage it.

There are also some other works that could be further studied with Picos++. As mentioned in the earlier chapters, It is always fast enough for the integrated system (judging from the task and dependence repetition rates). This means that the data communication and the software integration are the bottlenecks, therefore we could improve the integration and further boost the capability of a hardware accelerated runtime. In addition, we can make more usage of the Picos++ internal task-dependency graph management, and implement critical-path scheduling in hardware. Or with more threads in the future, we can try a variety of software scheduling policies for ready tasks in SMP and combine them with the hardware heterogeneous task scheduler for other hardware architectures in the system.

7.3 Financial Support

This thesis has been financially supported by the Spanish Government through Programa Severo Ochoa (SEV-2011-0067 and SEV-2015-0493), by the Spanish Ministry of Science and Technology through TIN2012-34557 and TIN2015-65316-P project, by the Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), by the European Research Council 7th FP, ERC RoMoL Grant Agreement number 321253, by the AXIOM project (ICT-01-2014 GA 645496), by the EuroEXA project (754337) and by the OmpSs on Android and Hardware support for runtime Project Cooperation Agreement with LG Electronics. We also thank the Xilinx University Program for its hardware and software donations.

Bibliography

[1] G. Al-Kadi and A. S. Terechko. A hardware task scheduler for embeded video processing. In *International Conference on High Performance and Embeded Architectures and Compilers(HiPEASC)*, 2009.

[2] O. ARB. OpenMP Application Programming Interface, version 4.5. [online], 2015. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[3] C. Architecture and P. S. L. C. in University of Delaware. The Codelet Execution Model Fine-Grain Multithreading for Extreme-Scale Computing. [online], 2017. http://www.capsl.udel.edu/codelets.shtml.

[4] R. A. Architecture. Riding on the Moore's Law (RoMoL) Workshop. [online], 2016. http://romol2016.bsc.es/.

[5] R. A. Architecture. Riding on the Moore's Law (RoMoL) Final Workshop. [online], 2018. https://www.bsc.es/romol/seminar.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15st International Conference on Parallel and Distributed Computing*, Euro-Par 2009, pages 863–874, 2009. Springer-Verlag.

[7] AVNET. Zedboard. [online], 2017. http://zedboard.org/.

[8] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, Mar. 2009.

[9] J. Balart, A. Duran, M. Gonzalez, X. Martorell, et al. Nanos Mercurium: A researchcompiler for OpenMP. *European Workshop on OpenMP (EWOMPâĂŹ04)*, 6:103–109, 2004.

[10] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the 2012 ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, 2012. IEEE Computer Society.

[11] N. Binkert, B. Bechmann, G. Black, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39:1–7, 2011.

[12] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '95, pages 207–216, 1995. ACM.

[13] J. Bosch. *Asynchronous Runtime for Task-Based Dataflow Programming Models*. PhD thesis, Universitat Politécnica de Catalunya, 2017.

[14] J. Bosch, X. Tan, et al. Characterizing and Improving the Performance of Many-Core Task-Based Parallel Programming Runtimes. *31st IEEE International Parallel and Distributed Processing Symposium Workshop*, 2017.

[15] J. Bosch, X. Tan, et al. Application Acceleration on FPGAs with OmpSs@FPGA. *The 2018 International Conference on Field-Programmable Technology (FPT)*, 2018.

[16] J. Bosch, X. Tan, et al. Asynchronous Task Creation for Task-Based Parallel Programming Runtimes. *The 2018 OpenMP Developers Conference (OpenMPCon)*, 2018.

[17] J. Bueno, L. Martinell, and A. Duran. Productive Cluster Programming with OmpSs. *International COnference on Parallel Processing(Euro-Par)*, 2011.

[18] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces. In *Proceedings of the 27th ACM International Conference on Supercomputing*, ICS '13, pages 359–368, 2013. ACM.

[19] D. Cabrera, X. Martorell, G. Gaydadjiev, E. AyguadÃl', and D. JimÃl'nez-GonzÃąlez. OpenMP extensions for FPGA accelerators. *International Symposium on Systems, Architectures, Modeling, and Simulation*, (17-24), 2009.

[20] D. Capalija and T. S. Abdelrahman. Microarchitecture of a Coarse-Grain Out-of-Order Superscalar Processor. In *International Transaction on Parallel and Distributed Systems*, 2013.

[21] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu. Runnemede: An Architecture for Ubiquitous High-Performance Computing. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, pages 198–209, 2013. IEEE Computer Society.

[22] M. Casas, M. Moreto, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. Unsal, A. Cristal, et al. Runtime-Aware Architectures. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing*, Euro-Par 2015, pages 16–27, 2015. Springer-Verlag.

[23] M. Casas, M. Moretó, et al. Runtime-Aware Architectures. In *Euro-Par*, pages 16–27. 2015.

[24] E. Castillo, L. Alvarez, M. Moreto, et al. Architectural Support for Task Dependence Management with Flexible Software Scheduling. *The 24th IEEE International Symposium on High-Performance Computer Architecture (HPCA'18)*, 2018.

[25] B. S. Center. BSC Application Repository(BAR). [online], 2014. https://pm.bsc.es/projects/bar/wiki/Applications.

[26] B. S. Center. Performance Tools. [online], 2016. http://www.bsc.es/computer-sciences/performance-tools.

[27] B. S. Center. OmpSs User Guide. [online], 2017. https://pm.bsc.es/ompss-docs/user-guide/.

[28] K. Chronaki, A. Rico, R. M. Bodia, et al. Criticality-aware dynamic task scheduling for heterogeneous system. *International Conference on Supercomputing (ICS)*, pages 329–338, 2015.

[29] K. Chronaki, A. Rico, M. Casas, et al. Task Scheduling Techniques for Asymmetric Multi-Core systems. *IEEE Transactions on Parallel and Districuted Systems (IPDPS)*, 28:2074–2084, 2017.

[30] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. *Workload Characterization. IEEE International Symposium*, (57-66), 2008.

[31] T. S. Czajkowski, U. Aydonat, D. Denisenko, et al. From OpenCL to high-performance hardware on FPGAs. *22nd International Conference on Field Programmable Logic and Applications (FPL)*, (531-534), 2012.

[32] T. Dallou, A. Elhossini, et al. Nexus#: A Distributed Hardware Task Manager for Task-Based Programming Models. In *IEEE 29th International Parallel and Distributed Processing Symp(IPDPS)*, 2015.

[33] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with Streams. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 35–42, 2003. ACM.

[34] R. H. Dennard, F. H. Gaensslen, H. nien Yu, V. L. Rideout, E. Bassous, Andre, and R. Leblanc. Design of Ion-implanted MOSFETs with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct. 1974.

[35] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 2011.

[36] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[37] N. Engelhardt, T. Dallo, et al. An Integrated Hardware-Software Approach to Task Graph Management. In *In 16th IEEE International Conference on High Performance and Communications(HPCC-2014)*, 2014.

[38] Y. Etsion, F. Cabarcas, et al. Task Superscalar: An Out-of-Order Task Pipeline. In *IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*, 2010.

[39] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory

Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, pages 83:1–83:11, 2006. ACM.

[40] V. Garcia, A. Rico, C. Villavieja, P. Carpenter, N. Navarro, and A. Ramirez. Adaptive Runtime-Assisted Block Prefetching on Chip-Multiprocessors. In *Proceedings of the 3rd International Workshop on On-chip Memory Hierarchies and Interconnets*, OMHI '14, pages 1888–1892, 2014. Springer-Verlag.

[41] Peter N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. White paper. 2009.

[42] M. GONZALEZ and J. BALART. Nanos Mercurium: A research compiler for OpenMP. *European Workshop on OpenMP*, 2004.

[43] J. Hoogerbrugge and A. Terechko. A multithreaded multicore system for embeded media processing. In *Transactions on High-performance Embeded Architectures and Compilers(THEA)*, 2011.

[44] IBM. Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems. [online], 2014. https://www.alpha-data.com/pdfs/capi_wp_29sept2014_pub.pdf.

[45] T. Instruments. INA219 Zero-Drift, Bidirectional Current/Power Monitor with I2C Interface. [online], 2015. http://www.ti.com/lit/ds/symlink/ina219.pdf.

[46] Intel. Intel Threading Building Blocks. [online], 2016. https://software.intel.com/en-us/intel-tbb.

[47] Intel. Intel FPGA SDK for OpenCL Programming Guide. [online], 2017. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf.

[48] A. Intel. Stratix 10 GX/SX Device Overview. [online], 2017. https://www.altera.com/en_US/pdfs/literature/hb/stratix-10/s10-overview.pdf.

[49] M. C. Jefferey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez. Data-centric execution of speculative parallel programs. *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, (1-13), 2016.

[50] M. C. Jefferey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. A Scalable Architecture for Ordered Parallelism. *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, (228-241), 2015.

[51] J. Kahle. The Cell Processor Architecture. In *Proceedings of the 38th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 38, page 3, 2005. IEEE Computer Society.

[52] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, 1993. ACM.

[53] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing with Streams. *IEEE Micro*, 21(2):35–46, Mar. 2001.

[54] S. Kumar, C. J. Hughes, et al. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *International Symposium on Computer Architecture*, 2007.

[55] S. Lee, J. Kim, and J. S. Vetter. OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing. *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 10(544-554), 2016.

[56] M. Manivannan, A. Negi, and P. Stenström. Efficient Forwarding of Producer-Consumer Data in Task-Based Programs. In *Proceedings of the 2013 42nd International Conference on Parallel Processing*, ICPP '13, pages 517–522, 2013. IEEE Computer Society.

[57] M. Manivannan and P. Stenstrom. Runtime-Guided Cache Coherence Optimizations in Multi-core Architectures. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 625–636, 2014. IEEE Computer Society.

[58] T. Mudge. Power: A First-Class Architectural Design Constraint. *IEEE Computer Journal*, 34(4):52–58, Apr. 2001.

[59] OpenMP Application Program Interface. Version 4.0. July 2013.

[60] V. Papaefstathiou, M. G. Katevenis, D. S. Nikolopoulos, and D. Pnevmatikatos. Prefetching and Cache Management Using Task Lifetimes. In *Proceedings of the 27th ACM International Conference on Supercomputing*, ICS '13, pages 325–334, 2013. ACM.

[61] P. K. Pearson. Fast Hashing of Variable-Length Text Strings. In *Communication of the ACM*, 1990.

[62] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta. Self-Adaptive OmpSs Tasks in Heterogeneous Enviroments. *The 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 138–149, 2013.

[63] 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. *IEEE Micro*, 30(5):16–29, Sept. 2010.

[64] J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2007.

[65] D. Sanchez, R. M. Yoo, et al. Flexible Architectural Support for Fine-Grain Scheduling. In *International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2010.

[66] SECO. AXIOM Board. [online], 2017. http://www.axiom-project.eu/2017/02/the-axiom-board-has-arrived/.

[67] M. Sjalander, A. Terechko, et al. A look-ahead task management unit for embeded multi-core architectures. In *Conference on Digital System Design(DSD)*, 2008.

[68] L. Sommer, J. Korinth, and A. Koch. OpenMP device offloading to FPGA accelerators. *IEEE 28th International Conference on Application-specific Systems, Architecture and Processors (ASAP)*, (201-205), 2017.

[69] StarPU. StarPU A Unified Runtime System for Heterogeneous Multicore Architecture. [online], 2017. http://starpu.gforge.inria.fr/.

[70] S. Subramanian, M. C. Jefferey, M. Abeydeera, et al. "Fractal: An execution model for fine-grain nested speculative parallelism". *International Symposium on Computer Architecture (ISCA)*, (587-599), 2016.

[71] X. Tan, J. Bosch, et al. Performance Analysis of a Hardware Accelerator of Dependence Management for Task-based Dataflow Programming models. *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 225–234, 2016.

[72] X. Tan, J. Bosch, et al. General Purpose Task-Dependence Management Hardware for Task-Based Dataflow Programming Models. *31st IEEE International Parallel and Distributed Processing Symposium*, 2017.

[73] X. Tan, J. Bosch, et al. Picos, A Hardware Task-Dependence Manager for Task-Based Dataflow Programming Models. *HPCS '17: Proceedings of the International Conference on the High Performance Computing and Simulation*, 2017.

[74] X. Tan, J. Bosch, et al. Hardware Heterogeneous Task Scheduling for Task-based Programming Models. *The 2018 OpenMP Developers Conference (OpenMPCon)*, 2018.

[75] X. Tan, J. Bosch, et al. Picos++: a Hardware Accelerated Runtime for Task-dependence Management and Heterogeneous Task Scheduling. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2018.

[76] X. Tan, J. Bosch, D. Jimenez-Gonzalez, C. Alvarez-Martinez, and E. Ayguade. Hardware Accelerator of Dependence Management for Task-based Programming Models. [online], 2016. http://acaces.hipeac.net/2016/index.php?page=home.

[77] X. Tan, J. Bosch, D. Jimenez-Gonzalez, C. Alvarez-Martinez, E. Ayguade, and M. Valero. Task Dependences Management Hardware Acceleration for Task-based Dataflow Programming models. [online], 2016. https://www.bsc.es/doctoral-symposium-2016/program.

[78] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems (IPDPS)*, 13:260–274, 2001.

[79] TU-Berlin. Starbench Benchmark Suite. [online], 2015. http://www.aes.tu-berlin.de/menue/forschung/projekte/abgeschlossene_projekte/starbench_parallel_benchmark_suite/.

[80] M. Valero, M. Moreto, M. Casas, E. Ayguade, and J. Labarta. Runtime-Aware Architectures: A First Approach. *International Journal on Supercomputing Frontiers and Innovations*, 1(1):29–44, June 2014.

[81] M. Valero, M. Moretó, et al. Runtime-Aware Architectures: A First Approach. *International Journal on Supercomputing Frontiers and Innovations*, 1, 2014.

[82] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Parallel Programming of General-purpose Programs Using Task-based Programming Models. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar '11, pages 13–13, 2011. USENIX Association.

[83] M. Vidal. *Synchronization/Communication techniques for OmpSs@FPGA*. PhD thesis, Universitat Politécnica de Catalunya, 2017.

[84] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architure News*, 23(1):20–24, Mar. 1995.

[85] XILINX. Zynq-7000 All Programmable SoC Overview. [online], 2017. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[86] XILINX. ZYNQ UltraScale+ MPSoC Overview. [online], 2017. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

[87] F. Yazdanpanah, C. Alvarez, D. Jimenez-Gonzalez, R. M. Badia, and M. Valero. Picos: A hardware runtime architecture support for OmpSs. *Future Generation Computer Systems(FGCS)*, 2015. http://dx.doi.org/10.1016/j.future.2014.12.010.

[88] F. Yazdanpanaha, D. Jimenez, et al. Analysis of the Task Superscalar Architecture Hardware Design. In *International Conference on Computational Science (ICCS)*, 2013.

[89] F. Yazdanpanaha, D. Jimenez-Gonzalez, C. Alvarez, R. Badia, and M. Valero. Picos: A Hardware Runtime Architecture Support for OmpSs. *The Future Generation Computing Systems*, 2014.

[90] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "Codelet" Program Execution Model for Exascale Machines: Position Paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, 2011. ACM.

List of Figures

List of Tables

Glossary

Picos General reference to our proposal, or the first prototype

Picos++ The second or third prototype

Plain A software-only runtime

HW Functional accelerator Specialized task execution unit

HwAccs HW functional accelerators

HwAcc tasks Tasks executed in HwAccs

HW tasks Tasks executed in HwAccs

GW Gateway module in Picos designs

TRS Task Reserve Station in Picos designs

TM Task memory in the Task Reserve Station in Picos designs

DCT Dependence Chain Tracker module in Picos designs

DM Dependence memory in DCT

VM Version memory in DCT

PDM Primer Dependence memory in DCT

FBM Fall-back memory in DCT

ARB Arbiter module in Picos designs

TS Task Scheduling module in Picos designs

TDG Task Dependency Graph

API Application Programming Interface

GPP General-purpose Processor

SMP Symmetric-Multicore Processor

SMP tasks Tasks executed in SMP

Heterogeneous tasks Tasks executed in heterogeneous hardware units

FPGA Field Programmable Gate Array

PL Programmable Logic

PS Processing System

APU Application Processing Unit

DMA Direct Memory Access

DRAM Dynamic Random-Access Memory

GPGPU General-Purpose Graphics Processing Unit

HPC High Performance Computing

ILP Instruction Level Parallelism

OS Operating System

RAM Random-Access Memory