Adicionando Suporte Nativo a Paralelismo de Tarefas a um Sistema RISC-V Multi-core com Suporte a Linux

Lucas Henrique Morais

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Mestrado em Ciência da Computação

Orientador: Prof. Dr. Alfredo Goldman

Coorientador: Prof. Dr. Guido Araújo

Este projeto foi apoiado pela FAPESP através dos processos 2017/02682-2 e 2018/00687-0

São Paulo, agosto de 2019

Resumo

Morais, L. H. **Adicionando Suporte Nativo a Paralelismo de Tarefas a um Sistema RISC-V Multi-core com Suporte a Linux**. Dissertação de Mestrado — Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2019.

Paralelismo por Tarefas é uma técnica genérica de extração de paralelismo de granularidade arbitrária aplicável a programas de vários domínios, com mínimo impacto sobre legibilidade de código, baseada na inferência automática de dependências de dados entre tarefas. O desempenho de aplicações paralelas baseadas nesse paradigma depende da velocidade com a qual o *runtime* de Paralelismo por Tarefas que lhe dá suporte é capaz de detectar tais dependências, fato que é ainda mais crítico para aplicações envolvendo tarefas de granularidade fina, já que nesse cenário o *overhead* de escalonamento não é amortizado por períodos significativamente maiores de computação útil. Recentemente, diversos grupos têm desenvolvido Sistemas de Suporte a Paralelismo por Tarefas acelerados por FPGAs, os quais são capazes de fazer *offload* das operações de inferência de dependências para um acelerador em FPGA de modo a melhorar o seu desempenho ao lidar com tarefas de granularidade fina.

Por outro lado, ainda que esses sistemas acelerados por FPGA apresentem ganhos substanciais com relação às alternativas baseadas puramente em software, o desempenho dessas soluções é prejudicado por gargalos de comunicação entre a CPU e a FPGA, os quais limitam a capacidade desses sistemas de lidar com cenários envolvendo grande número de núcleos ou tarefas muito finas. Motivados por isso, implementamos um Sistema de Suporte Nativo a Paralelismo por Tarefas — isto é, um processador com suporte arquitetural nativo a Paralelismo por Tarefas — com o objetivo de reduzir consideravelmente tais *overheads* de comunicação.

Mais especificamente, integramos a lógica em hardware do Picos, um acelerador de Paralelismo por Tarefas desenvolvido pelo Barcelona Supercomputing Center (BSC), ao Rocket Chip, uma implementação multi-core de código livre do RISC-V desenvolvida pela Universidade da Califórnia, Berkeley. O sistema resultante contém em sua ISA *(Instruction Set Architecture)* as instruções necessárias para que aplicações baseadas em tarefas possam interagir diretamente com essa lógica de escalonamento, minimizando os *overheads* associados ao uso de *runtimes* intermediários e eliminando toda a latência de comunicação FPGA-CPU.

Para avaliar a performance do protótipo que então se construiu, nós tanto (1) adaptamos o *runtime* de escalonamento de tarefas Nanos para que ele pudesse ser acelerado pelas novas instruções de escalonamento de tarefas, quanto (2) criamos um novo *runtime* leve de escalonamento de tarefas a que demos o nome de Phentos. Nossos experimentos mostram que programas baseados em paralelismo por tarefas usando o *runtime* Nanos-RV — a versão do *runtime* Nanos com suporte ao nosso sistema que produzimos — são executados em média 2,13 vezes mais rapidamente do que versões dos mesmos programas utilizando a versão básica do Nanos, enquanto programas executados com o Phentos são em média 13,19 vezes mais rápidos do que suas versões correspondentes baseadas na mesma versão básica do Nanos. Tais valores médios correspondem à média geométrica dos conjuntos de dados pertinentes. Usando oito núcleos, Nanos-RV entrega ganhos de desempenho com relação a execuções seriais de até 5,62 vezes, enquanto Phentos entrega ganhos de até 5,72 vezes.

Keywords: Paralelismo por Tarefas, Programação Paralela, RISC-V, Rocket Chip, Chisel.

Abstract

Morais, L. H. **Adding Native Support for Task Scheduling to a Linux-capable RISC-V Multicore System**. MSc Dissertation — Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2019.

The Task Scheduling Paradigm is a general technique for leveraging fine and coarse grain parallelism from applications of several domains with minimum impact on code readability, relying on the automatic inference of data dependencies among tasks. The performance of Task Parallel applications is correlated with the speed at which the underlying Task Scheduling System is able to detect such dependencies, something that is critical for fine-granularity workloads, which cannot amortize scheduling overheads with long periods of useful computation. That being the case, several groups have recently been developing FPGA-accelerated Task Scheduling Systems — architectures where a software Task Scheduling Runtime is able to offload its bookkeeping computations to an FPGA-based accelerator — with the goal of efficiently scheduling fine-grained tasks to CPU cores.

Even though these FPGA-accelerated systems offer substantial gains over the software-only baseline, it is also true that FPGA-CPU communication bottlenecks prevent such designs from handling scenarios with either large number of cores or very fine-grained tasks. With that in mind, we proposed the implementation of a Native Task Scheduling System — that is, a processor with native support for task scheduling embedded into its architecture — with the goal of substantially reducing these overheads.

More specifically, this project aimed at embedding the HW logic of Picos, a mature Task Scheduling Accelerator developed by the Barcelona Supercomputing Center (BSC), into Rocket Chip, an open-source, silicon-proven, multi-core implementation of RISC-V. The ISA of the resulting system provides special instructions for Task Applications to interact with this Task Scheduling Logic, ruling out all FPGA-CPU communication latencies.

To evaluate the prototype performance, we both (1) adapted Nanos, a mature Task Scheduling runtime, to benefit from the new task-scheduling-accelerating instructions; and (2) developed Phentos, a new HW-accelerated light weight Task Scheduling runtime. Our experiments show that task parallel programs using Nanos-RV — the Nanos version ported to our system — are on average 2.13 times faster than those being serviced by baseline Nanos, while programs running on Phentos are 13.19 times faster, considering geometric means. Using eight cores, Nanos-RV is able to deliver speedups with respect to serial execution of up to 5.62 times, while Phentos produces speedups of up to 5.72 times.

Keywords: Task Scheduling, Parallel Programming, RISC-V, Rocket Chip, Chisel.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For many of the parallel programming tools of today, easiness-to-use seems to come at the cost of performance. Consequently, writing correct and efficient parallel software in a productive way is a daunting task. While multi-core, heterogeneous systems have recently become commonplace — most smartphones now fall in this category —, software development tools for tapping their potential have not improved at the same rate.

Task Parallelism is a parallel programming model that allows a *Task Scheduling Runtime* to automatically schedule *tasks* to available processors while respecting the data dependencies between them. Runtime libraries implementing this model automatically infer data dependencies between tasks at execution time. The information necessary for doing so is provided by simple programmer annotations that indicate whether memory addresses pointed to by tasks are read from, written to, or both. Some have argued [Etsion et al., 2010] that Task Parallelism can be understood as a technique for semi-automatically tranforming sequential imperative programs into dataflow applications, for which elementary operations are performed concurrently and asynchronously, as soon as their input data become available. In fact, as Etsion et al. [2010] pointed out, task parallelism does for tasks the same that Tomasulo's Algorithm does for instructions — it infers data relationships between them and lets them run on an array of processing units according to their data dependencies.

Support for Task Scheduling with automatic inference of data dependencies was initially provided by OmpSS, OpenMP 4.0 and DepSpawn through their corresponding software runtime implementations [Balart et al., 2004, Duran et al., 2011, GNU Foundation, 2005, Intel Corporation, 2013]. Nonetheless, it was soon acknowledged that even though software-based Task Scheduling was good enough for extracting coarse grained parallelism, it could not cope with: (1) workloads involving very fine tasks; or (2) architectures with a large number of cores. In the first case, the long latency of the software runtime is comparable to the task execution time, thus impacting performance. In the second case, the insufficient scheduling throughput of the software runtime leads part of the cores to starve for work. In both cases, the overhead due to the runtime latency plays a central role in degrading system performance.

As a result, several research groups have sought to improve the maximum throughput of Task Scheduling systems by resorting to hardware accelerators (e.g. FPGA), leading to largely successful designs [Dallou and Juurlink, 2012, Tan, 2018, Tan et al., 2017, Yazdanpanah et al., 2015]. For example, Picos, the hardware task scheduler designed and implemented by the Barcelona Supercomputing Center (BSC) [Tan et al., 2017], was proven capable of significantly improving the performance of task parallel programs.

Yet, even though FPGA-accelerated Task Scheduling substantially expands the applicability of Task Parallelism with respect to software-only runtimes, CPU-FPGA communication overheads, low FPGA clock rates, and reliance on software drivers for accessing these accelerators lead to serious performance penalties that prevent such systems from handling the most demanding workloads.

In this work, we propose an architecture that integrates a Task Scheduling hardware accelerator into the processor core to minimize communication latency, reduce runtime overhead, and consequently increase the performance of applications parallelized with Task Scheduling.

1.1 Contributions

In light of what has been said so far, we prototyped a system where Task Scheduling functionality is not provided by an accelerator external to the CPU, but by logic sitting in the processor itself. By ruling out FPGA-CPU communication latencies, this architecture drastically reduces Task Scheduling overhead, meaning that tasks might be scheduled to cores at much higher rates. Designing this new architecture involved using the Chisel language [Bachrach et al., 2012] to integrate Picos, a mature Task Scheduling accelerator, to Rocket Chip, an open-source, silicon-proven, multi-core implementation of RISC-V [Asanović et al., 2016].

To allow this platform to have its performance measured with available Task Parallel applications, we adapted Nanos — a software-only Task Scheduling runtime targeting the OmpSs programming model — to our system. We also developed a new fly-weight, high-performance Task Scheduling runtime called *Phentos* from scratch, building upon the lessons learned while adapting Nanos.

Our prototype is also capable of running Linux, which was essential for the development of Nanos and Phentos, as they rely on threading libraries provided by the Linux environment. A description of how we managed to boot Linux on our system is provided in Appendix A.

As discussed in Section 6, the increased maximum task scheduling throughput displayed by our system gives it two closely related advantages over previous architectures (1) if mean task granularity is kept constant, it allows Task Scheduling programs to be efficiently scheduled to a larger number of cores, and (2) if the number of cores is kept constant, it allows workloads involving tasks more fine-grained than ever before to be efficiently executed. In fact, for workloads involving very fine-grained tasks, our system delivers application speedups of up to 146.01x (13.19x on average) with respect to a scenario where no Task Scheduling acceleration is used, with performance degradation (no larger than 3%) occuring for only 1 of the 37 analyzed workloads involving 5 different programs.

This dissertation is divided as follows: Section 3 presents our approach in the context of related work; Section 2 describes background material for the following sections; Section 4 details the proposed hardware architecture; Section 5 describes the software systems developed for letting the new architecture serve Task Scheduling applications; Section 6 discusses the experimental setup and its results; finally, we present our final remarks in Section 7. Some details on how our system has been implemented are described in Appendix A.

Chapter 2

Background

In this chapter, we provide an overview of several concepts and technologies that are relevant to the work reported by this dissertation.

2.1 Task Scheduling

The Task Scheduling Paradigm involves the scheduling of elementary computational units called *tasks* to processors according to the dependence relationships between them, an activity that is typically coordinated by a software Task Scheduling Runtime. According to the paradigm, task B is said to depend on some task A if, and only if, B is generated after A and one of the following propositions is true:

- Task A writes to some memory position p and B reads from p (RAW dependence)
- Task A writes to some memory position p and B writes to p (WAW dependence)
- Task A reads from some memory position p and B writes to p (WAR dependence)

2.2 RISC-V

RISC-V is an open and free RISC ISA that was initially developed by researchers from the University of California, Berkeley [Waterman et al., 2014]. It is currently managed by the RISC-V Foundation and has been used in several academic and commercial processor and microcontroller implementations. Current software support for it includes, for example, ports of GCC, Linux and QEMU. Development boards implementing RISC-V have been proven capable of running GPU-assisted applications on Linux environments. Some of its most well-known open-source implementations are BOOM [Celio et al., 2017], Ariane[1], and Rocket Core [Asanović et al., 2016].

2.3 Chisel

Chisel is a modern open-source HDL conceived by Berkeley researchers that is based on Scala [Bachrach et al., 2012]. Rocket Core and BOOM are among the most popular RISC-V processor implementations designed with this language.

[1] https://github.com/pulp-platform/ariane

2.4 Rocket Chip

Rocket Chip is a parametrizable, open-source, Chisel-based SoC generator of RISC-V systems capable of emitting synthesizable RTL code [Asanović et al., 2016]. It can be used for generating single-core or multi-core processors with either in-order (Rocket Core) or out-of-order (BOOM) pipelines. Caches, interconnects, and other system aspects are tailored by user-defined parameters.

2.5 Rocket Core

Rocket Core is a in-order open-source RISC-V implementation that can be instantiated in both 32-bit or 64-bit form. It supports easy integration of custom accelerators through its RoCC Interface.

2.5.1 RoCC Interface

This interface allows a compliant accelerator to make cache-coherent memory accesses and be exposed to user programs through custom instructions. The RoCC instruction format is described by Figure 2.1. There, fields `rs1` and `rs2` indicate the two optional operand registers; `rd` encodes the optional destination register; operands `xd`, `xs1`, and `xs2` indicate whether `rs1`, `rs2`, or `rd`, respectively, are used; `opcode` stores the instruction opcode; finally, `funct7` encodes the desired behavior, allowing instructions with identical opcodes to trigger distinct accelerator functionalities.

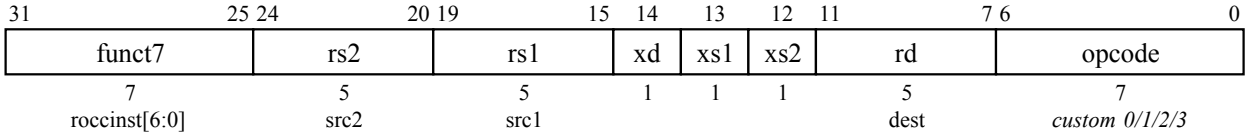

Figure 2.1: *Format of RoCC instructions.*

2.6 Terminology

This section defines some terms found in the rest of this work.

Task granularity refers to the duration of a task. We say that a task is *fine grained* if it has comparatively short execution time or, conversely, that it is *coarse grained* if it has comparatively large execution time.

Task retirement refers to the action by which a program informs a supporting Task Scheduling system that a certain task has finished executing.

In-flight task Refers to a task that is currently being executed.

Pending task Refers to a task that has been submitted but whose execution has not yet started.

Ready task refers to a task that does not depend on any in-flight or pending task and that is, consequently, ready to be executed.

Task submission refers to the action by which a program requests a supporting Task Scheduling system to add a new task to the task dependence graph.

Task waiting refers to the action by which a program blocks until a certain number of tasks have been retired.

Maximum Task Throughput is the number of tasks that a given task scheduling system is able to retire per unit of time, considering all scheduling overheads and assuming that task payloads are instantly executed by worker processors.

Chapter 3

Related Work

Researchers have been aware of the need for hardware acceleration of task-based systems since, at least, the late 2000s. The earliest solutions consisted on processor extensions for improving scheduling of dependence-less tasks. Then, as StarSs and then OpenMP 4.0 introduced tasks with data dependencies [Duran et al., 2008, Sinnen et al., 2007], new architectures were proposed for reducing task graph management overhead, and HDL implementations of several of these architectures were conceived [Dallou et al., 2015, 2016, Wang et al., 2015, 2013, Yazdanpanah et al., 2015].

Kumar et al. [2007] developed hardware task queues that could be used for accelerating the dynamic scheduling of tasks. It targeted workloads involving tasks whose inter-dependencies are either non-existing or explicitly enforced by the programmer using the fork-join model. Their work also aimed at improving load balance between cores by implementing a fast HW work stealing mechanism.

Meenderinck and Juurlink [2010] argued that the software StarSs runtime then available did not deliver enough scheduling throughput to let Task Parallel programs scale well in scenarios with more than five Cell cores, concluding that hardware acceleration was needed for those cases.

Etsion et al. [2010] proposed the Task Superscalar architecture, which aimed to improve the scheduling performance of tasks with dependencies. The design was evaluated with an adapted processor simulator fed with task parallel application traces from several domains.

Then, Dallou et al. [2015] brought about an FPGA-synthesizable hardware architecture called Nexus# for accelerating StarSs runtimes. This architecture was an optimized version of a prior HW task scheduler, Nexus++ [Dallou et al., 2013]. Although having similar goals to Task Superscalar, this work had the edge of encompassing an actual VHDL prototype that could be used for more precise performance evaluation.

Yazdanpanah et al. [2015] presented a similar HW task scheduler called Picos. After performing a thorough design space exploration of the main system components, this work presented performance results based on program traces and on timing analysis of a Picos VHDL implementation.

After that, Tan et al. [2016] extended the work on Picos by embedding it in the first Task Scheduling system capable of serving real Linux applications, based on a Zynq-7000 Xilinx development board. With this platform, Tan et al. [2016] confirmed that systems with HW-based Task Scheduling outperform software-only alternatives and showed that FPGA-CPU communication latencies have great impact on program performance. The Picos VHDL implementation used in that work is the same that we integrated to Rocket Chip.

Some time afterwards, Tan et al. [2017] unveiled Picos++, a new version of Picos with support

to nested tasks and improved FPGA-CPU communication.

Performance results presented by many of these works [Dallou et al., 2015, Wang et al., 2015, Yazdanpanah et al., 2015] do not assess the impact of communication latencies between the CPU and the HW Task Scheduler. Consequently, the performance figures they report are far more favorable than those reported for full-system implementations of Picos [Tan et al., 2016] or Picos++ [Tan et al., 2017]. Moreover, while these full-implementations succeed at proving that HW Task Scheduling consistently outperforms SW Task Scheduling, they have important limitations. First, they do not allow programs to be executed on more than two or four cores, since they are based on a dual-core or quad-core ARM+FPGA SoCs. Second, even though Picos++ greatly improves upon the original Picos FPGA-CPU communication scheme, the lifetime-cost of processing tasks[1] is in the range of thousands of processor cycles for any of these systems, degrading performance of fine-grained task applications.

Our system overcomes both of these limitations. Being based on Rocket Chip, it affords executions involving up to eight floating-point-enabled cores in our FPGA of choice, the ZCU102-ES2. Additionally, by embedding Task Scheduling logic into the processor itself, all FPGA-CPU communication latencies are eliminated, thus reducing the total amount of cycles needed for scheduling a task by up to two orders of magnitude with respect to the best previous system based on the same accelerator [Tan et al., 2017]. In so doing, it greatly expands the number of applications that can be efficiently handled by Task Scheduling.

[1] The total number of cycles taken by a Task Scheduling System to process and schedule a task during the whole lifetime of the latter, not considering the time spent running the useful computation encapsulated by the task.

Chapter 4

Picos + RISC-V Integration

4.1 Architecture Overview

Our work adds native Task Scheduling support to a Rocket Chip processor by integrating it with the Picos Task Scheduling accelerator. In order to do so, it introduces two major Chisel modules: *Picos Manager*, that is instantiated once in the system and is visible to all cores; and the *Picos Delegate* module, which implements the RoCC interface and that is instantiated once in every processor core. A simplified view of the architecture is given by Figure 4.1.

Task Scheduling functionality is provided by Picos and exposed to each core through custom instructions implemented by the core-specific RoCC *Picos Delegate* instances (called *ROCC Acc-Stub* in Figure 4.1). These instances will then interact with Picos through the mediation of *Picos Manager*, which implements logic for (1) ensuring the atomicity of some Picos-CPU transactions; (2) reducing the number of submission packets that must be provided by the CPUs by compressing null packets; (3) arbitrating the distribution of ready-to-run packets to cores; (4) allowing Chisel queues found in Rocket Chip to correctly interact with Picos queues, which implement a different handshake protocol; (5) buffering Picos-CPU transactions for hiding short Picos downtimes. In the near future, we plan to use *Picos Manager* as a platform for developing additional system optimizations such as task-scheduling-aware cache prefetching, faster parameter passing for tasks, etc.

4.2 The Software Interface

The main goal of this work was to develop a system with as little scheduling overhead as possible. To this effect, we not only leveraged the power of Picos to track task dependencies much faster than software runtimes but we also tried to keep communication latencies between Task Scheduling applications and Picos to a minimum. In our system, communication latencies are limited by the use of low-latency Picos-CPU dedicated datapaths bypassing system memory and by the provision of custom processor instructions for requesting Task Scheduling functionality. The existence of such instructions simplifies the construction of middleware to connect task applications to the underlying Task Scheduling hardware, thus avoiding additional software overheads.

While designing Picos Manager and the auxiliary Picos Delegate, we opted for making most of the new instructions non-blocking: of all instructions, only *Retire Task* is blocking. In this context, blocking instructions are those that only return after the corresponding transaction between Picos

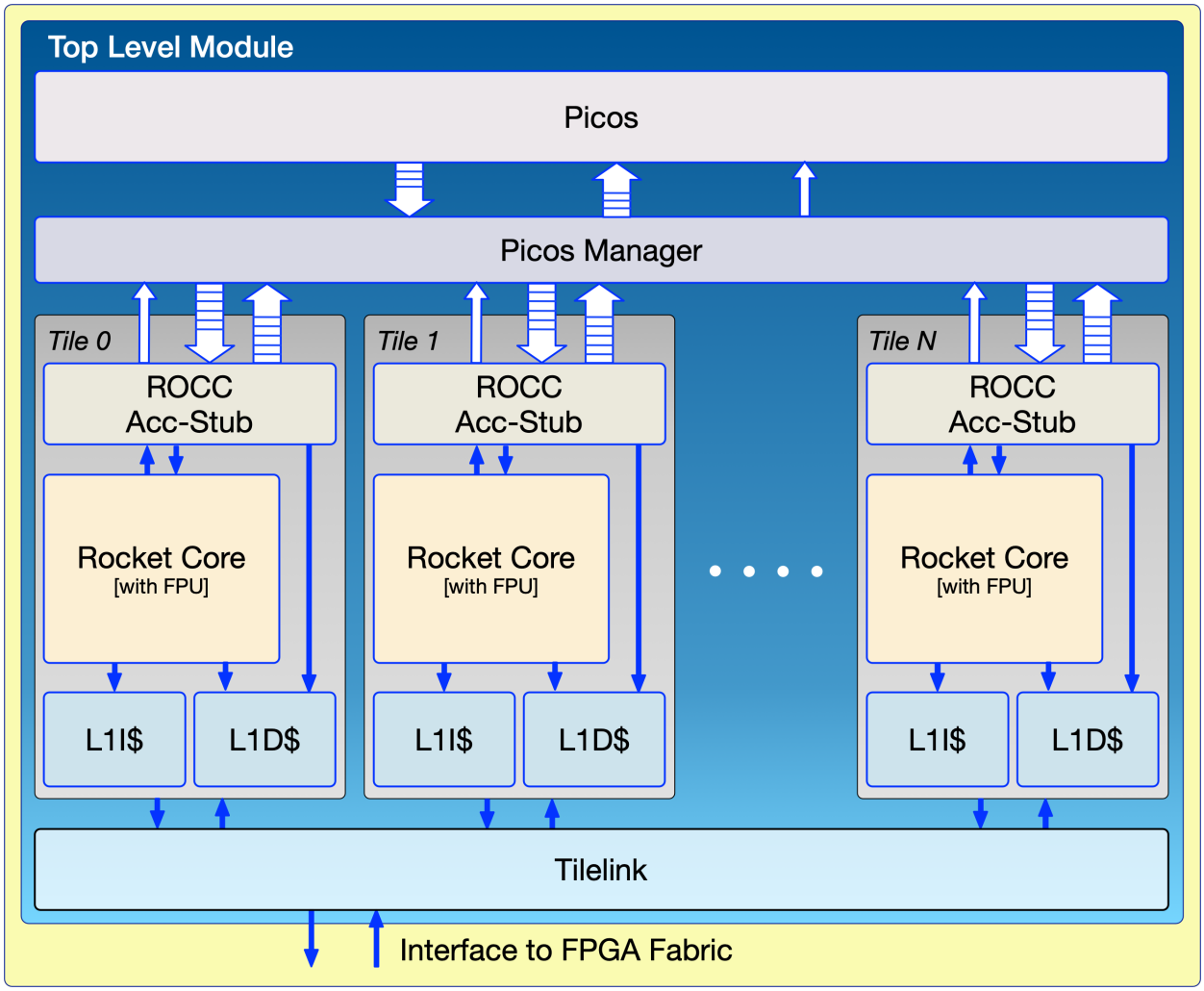

Figure 4.1: *Overview of the Picos + Rocket Chip system architecture*

Manager and the core executing the instructions has completed. Making most instructions non-blocking gives more freedom for the runtime/applications programmer to decide what to do in cases where Picos is not able to accept a new task or reply with a new ready task. If the system cannot complete the required actions, the related instruction returns a failure flag value and the program is free to keep trying. By quickly replying with these failure values, our system allows the runtime programmer to ask the core to sleep for a certain amount of time, saving energy; to perform alternative work actions; or even to request a context switch to the operating system. Additionally, having non-blocking instructions eases the development of deadlock-free systems, as we discuss in Subsection 4.3. On the other hand, the *Retire Task* instruction was designed as blocking because (1) Picos is always ready to receive new retirement signals, rendering the ability of a blocking version of the instruction to report failures back to the runtime useless, given that they will never occur; and because (2) this reduces compiler register pressure, as a blocking version of the instruction would require a result register to be available at the moment the instruction is executed.

All instructions implemented by the Picos Delegates are described by Table 4.1.

In a typical use scenario, as the execution of a task parallel application starts, some core c_i issues a *Submission Request* and, assuming the task has D dependencies, it executes *Submit Packet* $(3+3\cdot D)$ times — the number of packets needed for encoding a task with that many dependencies,

Name	funct	xs1	xs2	xd	Description
Submission Request	0x5	0	0	1	Informs the system that the core executing this instruction will attempt to submit a task.
Submit Packet	0x1	1	0	1	Submits a single 32-bit wide submission packet.
Submit Three Packets	0x7	1	1	1	Submits three 32-bit wide submission packets.
Ready Task Request	0x6	0	0	1	Requests the system to move a Ready Task packet from the global Ready Queue to the core-specific ready queue of the executing core.
Fetch SW ID	0x2	0	0	1	If the ready queue of the execution core is not empty, it returns the SW ID relative to the front element of the queue.
Fetch Picos ID	0x4	0	0	1	If the ready queue of the execution core is not empty and the SW ID relative to the front element of the queue has already been fetched, it returns the Picos ID of the same element and pops the queue.
Retire Task	0x3	1	0	0	Informs the system about the retirement of the task with a given Picos ID.

Table 4.1: *Custom Task Scheduling instructions supported by the system, following the RoCC instruction format described in 2.1.*

as Figure 4.4 shows. Then, some core c_j (possibly the same as c_i) issues a *Ready Task Request* to let its private ready queue be eventually filled by Picos Manager. Once Picos Manager has (1) noticed that a new Ready Task has been written to the global ready queue and that it has (2) answered all previous Ready Task Requests from any cores in their chronological order — a condition that is trivially satisfied by the fact that no other request has been previously issued — it pops data from the global ready queue and encodes it into a new entry of the private ready queue of core c_j. Then, when core c_j issues a *Fetch SW ID*, it is answered with the SW ID that core c_i provided to the system during task submission. Finally, after core c_j has finished executing the task, it issues a *Retire Task* to ensure that Picos removes the task from the Task Graph and possibly makes more tasks ready for execution.

4.3 Avoiding deadlocks by using non-blocking instructions

As previously mentioned, ensuring that submission and work-fetching instructions are non-blocking eases the development of deadlock-free systems. In the following lines, we present two scenarios where blocking instructions could lead to deadlocks and discuss ways to avoid them.

Deadlock Scenario 1: blocking submission instructions Let us suppose that some thread T might execute ready tasks and that it is the only allowed to submit new tasks to Picos. Let us also suppose that it successfully executes *Ready Task Request* while trying to fetch a new task but fails to get one by running *Fetch SW ID*. Finally, let us suppose that just after the latter instruction was executed, Picos Manager fills the core-specific ready queue of T with a new descriptor. Then, if for some reason T blocks while running any submission-related instruction (*Submission Request*, *Submit Packet*, or *Submit Three Packets*), it is possible that it will never recover from it.

This might happen because the two following facts: (1) submissions and submission requests might block when buffers and other internal data structures in either Picos or Picos Manager

become full; and (2) it is possible that more space might be available in these buffers and data structures only after the task descriptor now sitting in the core-specific ready queue of T is executed.

Consequently, if T blocks while performing a submission-related operation in a situation where it can only succeed after T consumes at least one element of its own core-specific ready queue, the whole system will stall.

Deadlock Scenario 2: blocking work-request As before, let us suppose that some thread T might execute ready tasks and that it is the only one allowed to submits tasks to Picos. Let us further suppose that just prior to the execution of *Ready Task Request* by T, the routing queue in the work-fetch arbiter (see Fig. 4.3) is full. In this case, the *Ready Task Request* instruction issued by T will block until writing to that routing queue is possible again. Nonetheless, if it is also true that there are no ready queue descriptors either on Picos or in Picos Manager's global *RoCC Ready Queue*, the routing queue will never be depleted before new ready-tasks are generated — since there are no ready tasks to distribute — and the *Ready Task Request* being executed by T will only return after new ready-tasks are made available. If the task graph managed by Picos does not contain any task waiting for the retirement of others to become ready, a new ready task will only be available after a new task submission succeeds, but a new submission can only take place after at least one ready task is fed to Picos Manager. Since these two events depend on each other, none of them will never happen, leading to a deadlock.

These deadlock scenarios can be avoided in several manners. In our system, we opted for making the submission and the work-fetching instructions non-blocking, which allows a thread holding the responsibilities of both generating and running tasks to freely switch between these roles. Alternatively, one could have decided to keep these instructions blocking and, for example, use atomic shared variables to ensure that threads with multiple roles never blocked after performing actions related to role R_1 while it still had pending actions regarding role R_2. This approach should lead to higher software complexity and slightly lower performance due to the atomic memory transactions required, nonetheless.

4.4 Picos

Picos is the module responsible for providing fast Task Scheduling functionality. Its communication interface includes queues for (1) receiving information about new tasks to be added to the task graph, called submission queue; (2) informing the outside world about tasks that are ready to be executed, called the ready queue; (3) being informed that a task has retired, called the retirement queue.

More details about Picos might be found in related publications [Tan, 2018, Tan et al., 2017, Yazdanpanah et al., 2015].

4.5 Picos Delegate

The ISA extension interface defined by our architecture is implemented by the Picos Delegate instantiated in every core. The following lines describe how this accelerator implements each of the supported custom instructions.

4.5.1 Submission Request

Before issuing submission packets, software running at a given core should issue a *Submission Request* describing the number of packets to be submitted. This serves two purposes: first, it makes sure that submission packets coming from such core will be forwarded to Picos before packets relative to later submissions coming from other cores; second, it allows the system to infer how many zero-packets should be sent to Picos after the non-zero packets, considering that Picos always expects to receive 48 32-bit packets. The non-zero packets of a task with D data dependencies should be followed by $48 - (3 + 3 \cdot D) = 45 - 3 \cdot D$ zero packets (see Figure 4.4). Such null packets are automatically generated by Picos Manager after the relevant Picos Delegate sends the last non-zero packet.

4.5.2 Submit Packet

The Picos Delegate fulfills *Submit Packet* instructions by simply forwarding the lower 32-bits of their single register operand to Picos Manager, which will then be responsible to forward the packet to Picos.

4.5.3 Submit Three Packets

The *Submit Three Packets* instruction is a variation of *Submit Packet* that submits three 32-bit packets at a time. The three submission packets $P1$, $P2$, and $P3$ are retrieved from the `rs1` and `rs2` operand registers, with $P1 = rs1(63, 32)$, $P2 = rs1(31, 0)$, and $P3 = rs2(31, 0)$. This instruction is useful for reducing the amount of cycles taken for submitting tasks. Given that the number of non-zero packets of Picos task descriptors is always a multiple of three, task submissions may be accomplished without resorting to the simpler and slower 1-packet version of this instruction.

4.5.4 Ready Task Request

Picos Delegates do not have direct access to the single ready queue of Picos. Rather, each of them is allowed to pop contents of its core-specific ready queue inside Picos Manager. On the other hand, Picos Manager only forwards ready packets from Picos to these private queues after being requested to do so. Picos Delegates issue such requests upon the decoding of *Ready Task Request* instructions. After receiving such request R from a core c_i with ready queue q_i, Picos Manager is guaranteed to only answer later ready task requests by any core after having satisfied R. Consequently, Picos Manager distributes ready-to-run tasks in the same order that ready task requests come from the cores.

4.5.5 Fetch SW ID

Suppose that core c_i, with private ready queue q_i, issues a *Fetch SW ID* instruction. If q_i is empty, the Picos Delegate instance in that core fulfills the instruction by returning a failure value; otherwise, it fulfills the instruction by returning the SW ID encoded by the front element of the queue and setting an internal flag signaling its success. In either case, it does not pop q_i.

4.5.6 Fetch Picos ID

Suppose that core c_i, with private ready queue q_i, issues a *Fetch Picos ID* instruction. If, and only if, q_i is not empty and a previous *Fetch SW ID* instruction succeeded at retrieving the SW ID encoded by the front element of q_i, it fulfills the instruction by returning the Picos ID encoded by the front element, popping q_i, and resetting the internal flag marking the success of a previous *Fetch SW ID* instruction. If q_i is empty and/or a previous *Fetch SW ID* instruction has not succeeded at retrieving the front element of q_i, *Fetch Picos ID* will return a failure value and will not change any internal state of Picos Manager.

4.5.7 Retire Task

The Picos Delegate fulfils *Retire Task* instructions by pushing the payload of the operand register to the Work-Fetch Arbiter in Picos Manager (see Figure 4.3). Even though this operation is blocking — the *Retire Task* instruction only succeeds after the arbiter-mediated transaction has finished — cores will most frequently be able to write the retirement packet right away, given that Picos consumes retirement packets fast enough for making sure that its internal retirement queue can always receive a new packet from the serializing Work-Fetch Arbiter.

4.6 Picos Manager

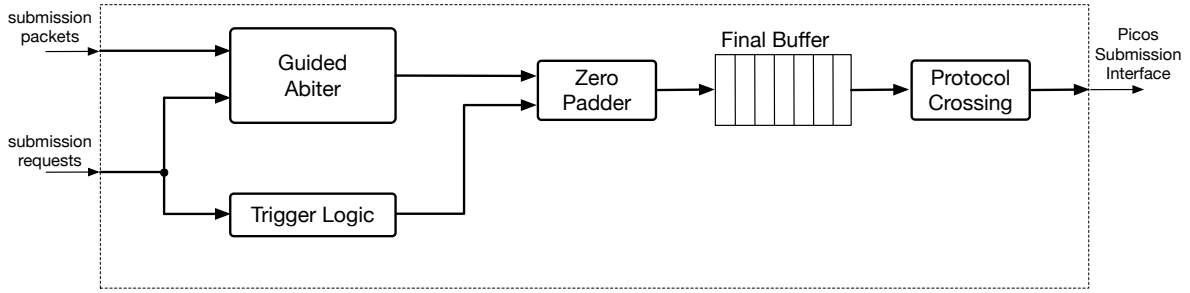

Figure 4.2: *Block diagram of the Submission Handler, a module instantiated by Picos Manager for carrying out transmission of new task descriptors to Picos.*

Picos Manager allows communication between Picos and the core-specific Picos Delegates without modification of the Picos interface. Additionally, it improves system performance by converting compact submission packet sequences, which come from the Picos Delegates and can have as few as three packets, to Picos-compliant submission packet sequences, which are always 48 packets long.

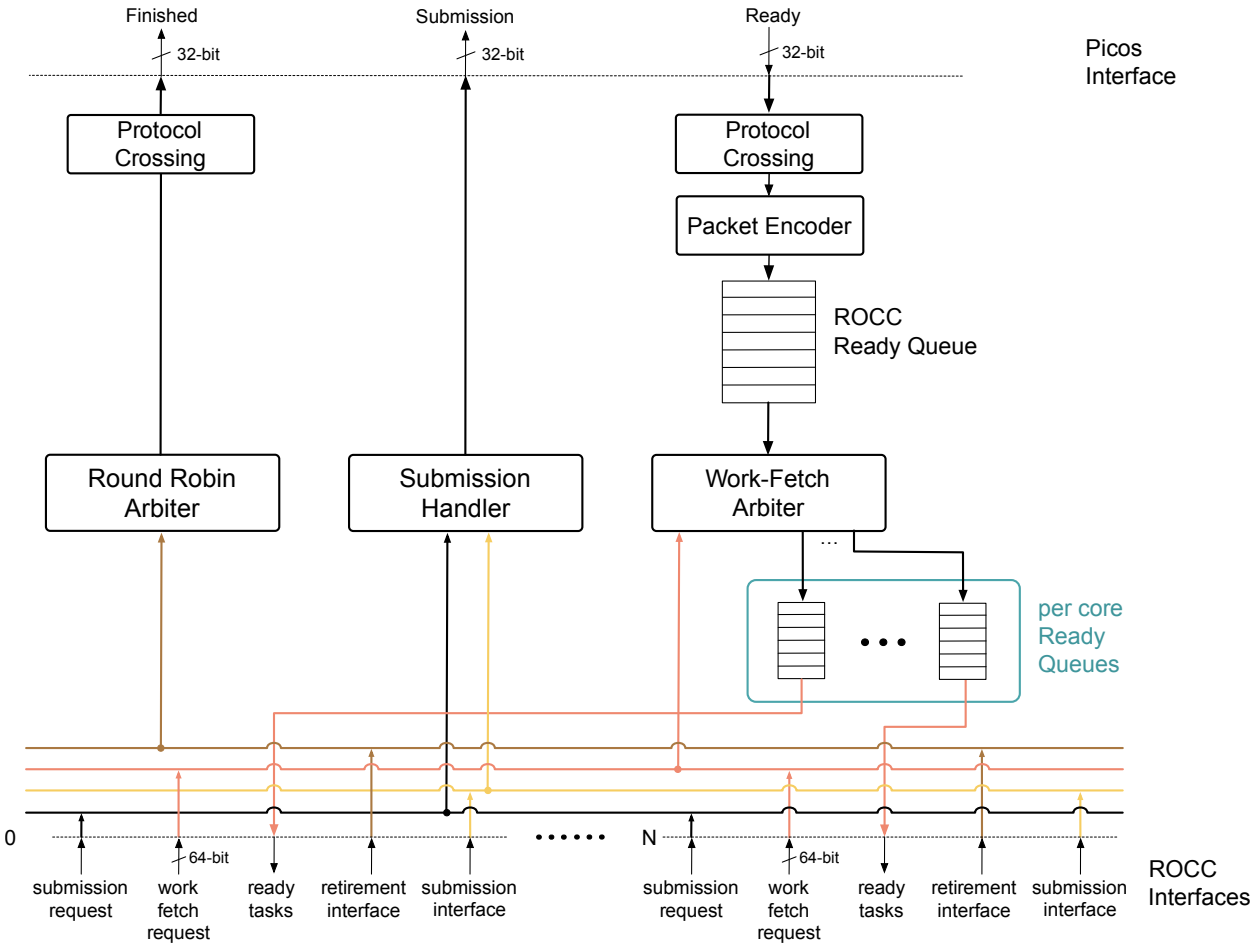

Figure 4.3: *Internals of the Picos Manager module.*

4.6.1 Interface

As shown by Figure 4.1, Picos Manager is connected to Picos and each of the core-specific Picos Delegates. Its core-specific interface, which is replicated for each core, includes (1) a ready queue, (2) a retirement queue, (3) a submission queue, (4) a submission request queue, and a (5) work fetch request queue; its Picos-facing interface includes (6) ready, (7) retirement, and (8) submission queues; finally, its debug interface (omitted for simplicity from Figure 4.1) includes (9) a 4-bit output signal encoding errors (omitted from figures).

4.6.2 Structural elements

As described by Figure 4.3, Picos Manager comprises the following main elements: the Round Robin Arbiter, the Submission Handler, the Work-Fetch Arbiter, several protocol crossing modules, a Packet Encoder, and core-specific ready queues. In the following lines, we discuss their behavior and implementation.

Submission Handler The submission handler — shown in detail in Figure 4.2 — is the module that handles processing of submission packets in behalf of Picos Manager. It serves three main purposes: (1) making sure that submission packet sequences coming from cores are not interleaved, given that Picos requires task submissions need to happen atomically; (2) enabling faster submission operation by automatically padding non-zero submission packet sequences with the necessary zero packets for completing the 48-packets-long sequences expected by Picos; (3) implementing the protocol crossing logic necessary for allowing the standard Chisel queues employed in Rocket Chip to adequately interact with the submission interface of Picos.
For achieving these goals, the Submission Handler relies on the Guided Arbiter, which makes sure that, at any given moment, (1) only one core is allowed to transmit submission packets to Picos and that (2) access to Picos submission interface is transfered between cores only after the last transmission sequence to Picos has been completed. Finally, zero-padding of packet sequences is implemented by a Zero Padder module.

Work-Fetch Arbiter The Work-Fetch arbiter is responsible for distributing ready-to-run task descriptors to cores according to the total-order at which they requested such data. We implemented the Guided Arbiter using an *InOrderArbiter* — available in the Rocket Chip source tree as a stock library module — and some additional low-abstraction logic.

Protocol crossing modules These modules allow the standard Chisel queues employed in Picos Manager to correctly interface with the Task Retirement and Ready-Task interfaces of Picos. They should, for example, ensure that Picos queues and standard Chisel queues correctly interact in spite of the former being non-fallthrough while the latter are fallthrough. Figures 4.9, 4.7 and 4.8 present typical waveforms for full procotol-crossing-mediated submission, work-fetching, and retirement transactions. Having a clear picture of the communication protocols for interacting with Picos queues is essential for developing a bug-free system employing it. In fact, a good portion of our debugging time was spent making sure that communication with Picos through its queues happened flawlessly.

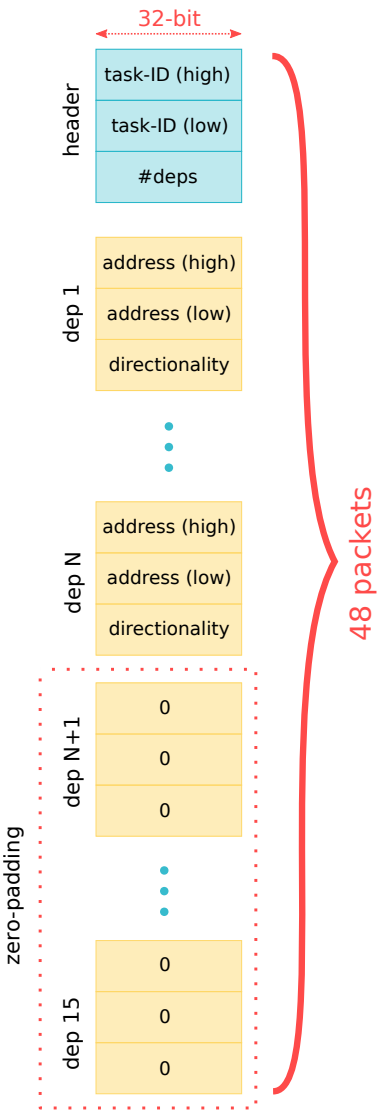

Figure 4.4: *Picos encoding of a task with* N *data dependencies. This shows that any task is described by* $3 * (15 + 1) = 48$ *packets. For a task with* N *data dependencies, the last* N_z *of these packets will be equal to zero, where* $N_z = (15 - N) * 3$. *In our system, only the first* $48 - N_z$ *packets must be submitted by the Task Scheduling Runtime to Picos Manager, as the* N_z *zero packets will be automatically appended by the Submission Handler. Only tasks with up to 15 data dependencies are supported.*

Packet Encoder This module compresses the three 32-bit ready-task encoding packets produced by Picos for every ready-to-run task into a single 96-bit packet (described in Figure 4.5), which is then stored in a central Ready-Task queue. It also verifies the success of the ready task transactions from Picos to Picos Manager by leveraging a Picos-provided XOR checksum. This module expects to receive from Picos through the Protocol Crossing in the format outlined by Figure 4.6.

Round Robin Arbiter This is a standard Chisel module that arbitrates multiple-producer-single-consumer connections in a round-robin fashion. It is used for merging the task retirement signals coming from different cores into the single retirement interface of Picos.

Core-specific ready queues These are buffers that hold 96-bit *(Picos ID, SW ID)* tuples (depicted in Figure 4.5) describing ready-to-run tasks. Picos Manager contains one instance of such buffer for every core in the system. Having such buffers ensures that, in scenarios with plenty of ready-to-run tasks, half of the 8-cycle long latency for fetching from Picos the three 32-bit packets describing a ready-to-run task is hidden from the application, which will be able to fetch the corresponding 96-bits of data with two 2-cycle-long RoCC instructions (*Fetch SW ID* and *Fetch Picos ID*).

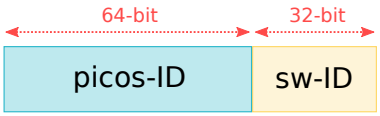

Figure 4.5: *In Picos Manager, both the global Ready Queue — which holds data coming from Picos Ready Queue — and the core-specific Ready Queues have entries like the (`picos-ID`, `sw-ID`) tuple described in this figure.*

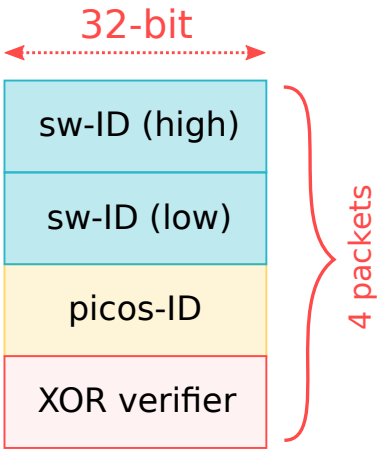

Figure 4.6: *The four packets provided by Picos for describing a single ready task. Packets are transmitted in top-to-bottom order. The last packet, the* XOR verifier*, corresponds to the XOR value of the three previous packets, and is used for checking the integrity of the transaction.*

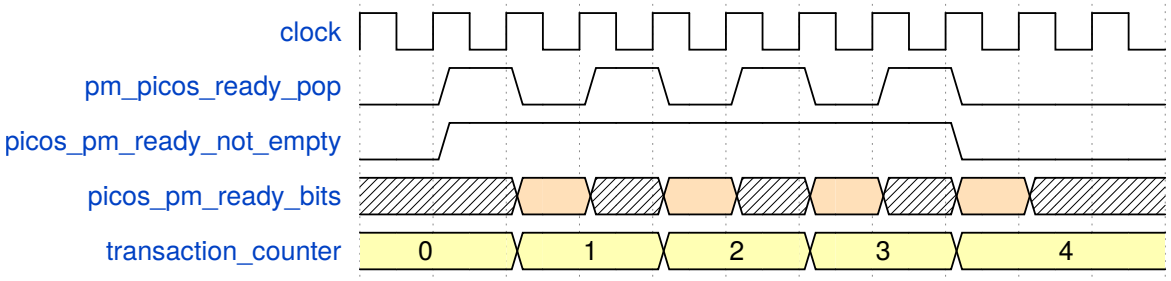

Figure 4.7: *Possible waveform for a full ready task transaction between Picos and Picos Manager. It describes how Picos expects from its ready queue to occur. In this figure, one can see that data payloads are only made available in the next cycle after Picos Manager has held its `pm_picos_ready_pop` signal high. This manifests the non-fallthrough nature of Picos' ready task queue.*

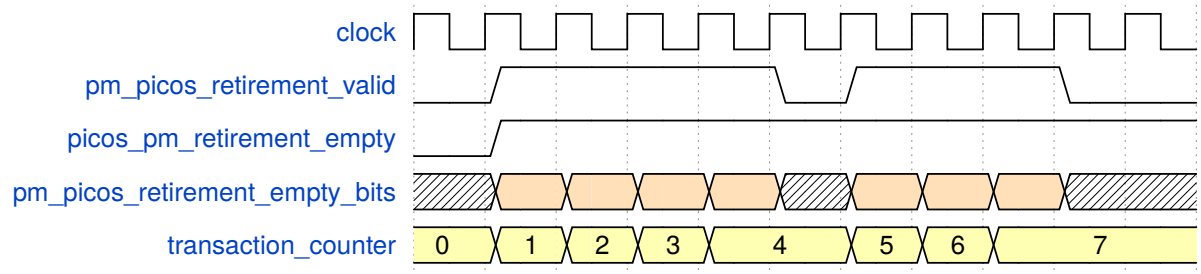

Figure 4.8: *Waveform describing system behavior for a possible scenario in which seven retirements occur in quick succession. It describes how Picos expects writes to its retirement queue to happen. A retirement transaction occurs at every cycle during which* `pm_picos_retirement_valid` *and* `picos_pm_retirement_empty` *are simultaneously high.*

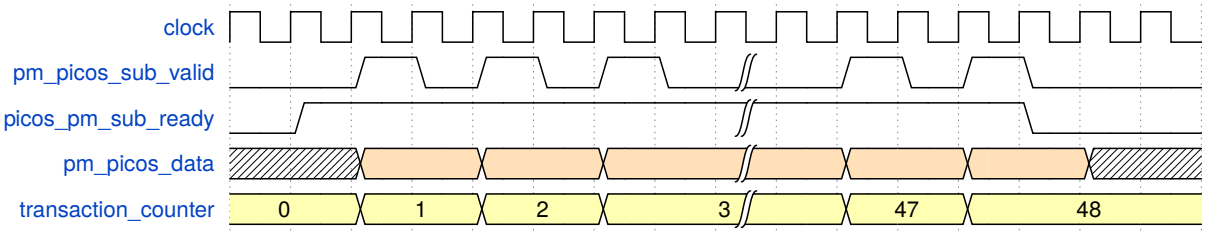

Figure 4.9: *Waveform describing system behavior for a possible scenario in which the 48 submission packets relative to a single task are exchanged between Picos and Picos Manager. It describes how Picos expects writes to its submission queue to happen. A transaction occurs at every cycle during which* `pm_picos_sub_valid` *and* `picos_pm_retirement_ready` *are simultaneously high.*

Chapter 5

Developing Tightly-Integrated Task Scheduling Runtimes

With the purpose of evaluating the performance of our system with Linux-based Task Scheduling runtimes, we both (1) ported to our system Nanos-SW, a mature Task Scheduling runtime targeting the OmpSs programming model, and (2) created Phentos — a light-weight, high-performance task scheduling runtime — from scratch. The first of these endeavors was useful for showing that our system is capable of running any OmpSs-complying Task Parallel application not generating nested tasks (which are not supported by the iteration of Picos integrated by our system). On the other hand, developing a new minimal runtime gave us the opportunity to avoid several sources of SW overhead that were identified in Nanos and that could not be easily removed without a major library re-write. In the following lines, we will discuss how each of these runtimes were built and how the tightly-integrated Task Scheduling accelerator contributed to their improved performance with respect to Nanos-SW.

5.1 Building Nanos-RV from Nanos-SW

Nanos is a software runtime supporting the OmpSs programming model that is maintained by the Barcelona Supercomputing Center. It was designed to easily accommodate new features as *plugins* that are dynamically linked to its core system depending on environmental variables or command-line arguments. It served as an important testing ground for the Task Parallel constructs based on automatically-inferred data dependencies that were introduced by the OmpSs model, helping them be integrated to OpenMP 4.0.

Using the plugin interface of Nanos, we developed a new Nanos module capable of offloading data-dependence-inference computation to Picos using the custom instructions implemented by our architecture. When this plugin is activated by setting the `NX_ARGS` environment variable as `NX_ARGS="-deps=picos"`, our custom instructions are used for submitting task descriptors to Picos, fetching ready-to-run tasks, or informing Picos of retiring tasks. The new *picos* dependence-inferring plugin replaces the default *plain* plugin, which achieves the same through software. In this work, we refer to Nanos using the *plain* plugin as Nanos-SW and to Nanos using the *picos* plugin as Nanos-RV.

Nanos modularity comes with a price, though. The plugin interface relies heavily on virtual func-

tions for implementing policy-oriented design, causing extra memory accesses for task submissions, retirements, and work-fetches. Additionally, Nanos code makes heavy use of mutexes and conditional variables for coordinating accesses to its shared data structures, leading to the performance penalties of the related system calls. Moreover, the ready-to-run tasks identified by Picos are not immediately scheduled to the core that fetched the corresponding ready-task descriptors, but are redirected to a Scheduler singleton that pushes all descriptors fetched from all cores through a single ready-task queue that all cores will then be allowed to access — which is much more inefficient.

Even though the results of Section 6 show that Nanos-RV is substantially faster than Nanos-SW, the overhead issues just discussed motivated us to implement a more light-weight runtime that could allow us to enforce stronger optimizations targeting the specificities of the new architecture.

5.2 The Phentos Fly-Weight Runtime

In this section, we describe the goals that guided Phentos design (Subsection 5.2.1) and discuss the two APIs that might be used for allowing task parallel applications to use it. Subsection 5.2.3 describes the ORD-Phentos API, while Subsection 5.2.5 describes FAST-Phentos. Their execution models are detailed in Subsections 5.2.2 and 5.2.4, respectively.

5.2.1 Overview

Phentos was designed with the following goals:

1. avoiding all non-IO syscalls, including those related to mutexes and conditional variables;
2. minimizing the number of cache-line invalidations per submission event;
3. minimizing the number of cache-line moves per work-fetching event;
4. minimizing function-call overhead by making most runtime API methods inlinable in application code;
5. mitigating the cache-bouncing problem by minimizing writes to atomic shared variables;
6. avoiding false-sharing with cache-aware data packing.

Phentos is implemented as a header-only (hpp) library, in a similar spirit to that of the amply used Boost library. This allows Phentos API methods be inlined in application code by the compiler, complying with design goal (4).

In Phentos, the only data structures that are shared between threads are the *Task Metadata Array* — an array of task metadata descriptors and — a single atomic counter of task retirements — which is necessary for implementing the `taskwait` construct used by many Task Parallel applications. For avoiding false-sharing and contributing to the minimization of cache-line invalidations per submission event (goals 2 and 6), the *Task Metadata Array* is implemented in such a way that the size of each of its elements corresponds to either one or two cache lines (sufficient for representing seven or fifteen task dependencies, respectively). A pre-processor macro controls which of the two element sizes is used, according to the needs of the particular Task Parallel application employing Phentos.

Also, Phentos is designed in such a way that any active element of the *Task Metadata Array* will only be accessed by the single thread that holds the `sw-ID` corresponding to such element. Threads obtain such identifiers by running *Fetch SW ID* instructions as described in Section 4. Consequently, the mere fact that two threads will never compete for the same data elements deems the use of synchronization artifacts like mutexes and conditional variables unnecessary in this context, contributing to design goal (1).

As it is widely known, having a spin-locked thread frequently verifying a memory position that is also frequently updated by other threads leads to the so-called cache-line bouncing problem. Combining memory accesses in such a way causes cache traffic between the relevant cores be dominated by the operations needed for transmitting data between the memory-modifying cores and the one monitoring changes. As a result, the general performance of all participating cores quickly degrades, as less cache-interconnect bandwidth is available for other memory accesses. Additionally, individual updates to the contended line might also take much longer to complete, as writers need to fetch updated content from other writers before performing their own update. Such problem is specially problematic for multi-core systems implementing the MESI coherence protocol — as it is the case for the Rocket Chip platform we based our prototype on — given that this protocol does not allow dirty cache lines to be directly communicated between caches. Instead, these systems require dirty cache lines to be communicated through the one-step higher cache level, which might even be main memory.

In order to avoid spin-locks from causing this problem, several strategies might be employed. The one implemented by Phentos for minimizing contention for the single atomic counter of task retirements involves (1) letting cores keep a private retirement counter that it might freely update; (2) only allowing cores to update the shared atomic counter if their private counters are non-zero and a given number of *Fetch SW ID* failures happened since the last shared counter update it performed; (3) making the spin-locking thread monitoring the atomic counter check the variable only after every N cycles. By doing so, it fulfills design goal (5).

Finally, the compact single or double cache-line task metadata representation employed by Phentos allows the task metadata of ready-tasks be fetched with only one or two cache line transfers, contributing to design goal (3).

5.2.2 ORD-Phentos execution model

The ORD-Phentos API defines the most general way of parallelizing an application with Phentos. Even if FAST-Phentos might deliver higher performance than ORD-Phentos, some applications are not amenable to be parallelized with the former, as we shall discuss in Subsection 5.2.4. In the remaining of this Subsection, we describe how Phentos handles task submission, fetching, execution, and retirement when ORD-Phentos is used.

Phentos initialization Once Phentos is initialized, worker threads are created and Phentos internal variables are set to default values. Worker threads will continuously try fetching and executing ready tasks until the program finishes.

Submission Picos expects Phentos to maintain information about the function pointer and parameters of in-flight tasks, given that Picos describes ready tasks simply by the related 64-bit `SWID` and `HWID` provided by Phentos during task submission. Consequently, before a task

is submitted, the Phentos-based application should add information about that task to the `metadataArray`, which will later be used for retrieving information about ready tasks coming from Picos. This is exemplified by lines 14 to 21 of Listing 1. The `metadataArray` should be indexed by the unique `SWID`s of in-flight tasks. The application might get a `SWID` for a new task by simply calling the `getNewSWID()` function as exemplified by line 13 of Listing 1 and described in Subsection 5.2.3.

Once the application has written the task metadata to the `metadataArray`, it might make a submission request and later start submitting task packets, as exemplified by lines 28 to 34 of Listing 1.

Work-fetching and Execution In a system with N cores, Phentos will initialize $N-1$ worker threads during its initialization. All such threads continuously try fetching and executing ready tasks. Additionally, several blocking functions that might be called by the submission thread allow tasks to be fetched and executed while the unblocking condition is not reached.

When a work-fetching action is successful, Phentos finds out the `SWID` and the `HWID` of a ready task. The function pointer of the ready task is retrieved by ORD-Phentos by consulting the `metadataArray`.The functions encapsulated by ORD-Phentos tasks should take their `SWID` as their only parameter. Once they are called by the worker thread, such functions might then use the `SWID` to reconstruct its data dependencies from the `metadataArray`, as exemplified by lines 2 to 6 of Listing 2.

As soon as these data dependencies are available, the function might execute its payload as usual. If profiling is enabled by the `ENABLE_PROFILING` macro, ORD-Phentos will evaluate the execution time of every task and add such value to the `totalExecutedTaskCycles` atomic variable.

Retirement Once a task is retired, ORD-Phentos recycles its corresponding `metadataArray` entry, informs Picos that the task has finished executing — possibly making more tasks ready —, and increments its atomic `numRetiredTasks` counter.

Task wait As described in Section 2.6, task waiting is the action by which a program blocks until a certain number of tasks have been retired. By keeping track of the number of all tasks that have been retired by any thread using the `numRetiredTasks` shared counter, Phentos allows this functionality to be implemented by simply busy-waiting until that counter has reached some given number of tasks. This is the behavior of the `task_wait()` function implemented by both ORD-Phentos and FAST-Phentos.

Instead of just waiting for the task wait condition to be reached, though, both ORD-Phentos and FAST-Phentos provide alternative task waiting functions that execute tasks while waiting for the target number of task retirements to be reached. ORD-Phentos offers this functionality through its `task_wait_and_try_executing_tasks()` function. Its use is exemplified by line 44 of Listing 1.

```
void bs_thread(void * tid_ptr, fptype * prices) {
  int i, j;
  fptype priceDelta;
  int tid = * (int * ) tid_ptr;

  unsigned long long swID;
  unsigned long long lastSWID = 0;
  unsigned long long num_iterations = 0;
  unsigned numPendingWorkRequests = 0;

  for (j = 0; j < NUM_RUNS; j++) {
    for (i = 0; i <= (numOptions - BSIZE); i += BSIZE) {
      swID = getNewSWID(lastSWID);
      metadataArray[swID].functionAddr = (unsigned long long) FA_BlkSchlsEqEuroNoDiv;
      metadataArray[swID].depAddresses0[0] = (unsigned long long) & sptprice[i];
      metadataArray[swID].depAddresses0[1] = (unsigned long long) & strike[i];
      metadataArray[swID].depAddresses0[2] = (unsigned long long) & rate[i];
      metadataArray[swID].depAddresses0[3] = (unsigned long long) & volatility[i];
      metadataArray[swID].depAddresses0[4] = (unsigned long long) & otime[i];
      metadataArray[swID].depAddresses0[5] = (unsigned long long) & otype[i];
      metadataArray[swID].depAddresses0[6] = (unsigned long long) & prices[i];

      asm volatile("fence"::: "memory");

      make_submission_request_or_work(24, 0, numPendingWorkRequests);
      submit_three_or_work(swID, 7, numPendingWorkRequests);

      submit_three_or_work((unsigned long long) & sptprice[i], 0, numPendingWorkRequests);
      submit_three_or_work((unsigned long long) & strike[i], 0, numPendingWorkRequests);
      submit_three_or_work((unsigned long long) & rate[i], 0, numPendingWorkRequests);
      submit_three_or_work((unsigned long long) & volatility[i], 0, numPendingWorkRequests);
      submit_three_or_work((unsigned long long) & otime[i], 0, numPendingWorkRequests);
      submit_three_or_work((unsigned long long) & otype[i], 0, numPendingWorkRequests);
      submit_three_or_work((unsigned long long) & prices[i], 1, numPendingWorkRequests);

      lastSWID = swID;

      num_iterations++;
    }
    BlkSchlsEqEuroNoDiv( & sptprice[i], & strike[i], &
      rate[i], & volatility[i], & otime[i], &
      otype[i], 0, & prices[i], numOptions - i);

    task_wait_and_try_executing_tasks(num_iterations);
  }
}
```

Listing 1: *Task-spawning code from the Blackscholes benchmark, parallelized with the ORD-Phentos API.*

```
void FA_BlkSchlsEqEuroNoDiv(unsigned long long swID) {
  fptype * sptprice = (fptype * ) metadataArray[swID].depAddresses0[0];
  fptype * strike = (fptype * ) metadataArray[swID].depAddresses0[1];
  fptype * rate = (fptype * ) metadataArray[swID].depAddresses0[2];
  fptype * volatility = (fptype * ) metadataArray[swID].depAddresses0[3];
  fptype * time = (fptype * ) metadataArray[swID].depAddresses0[4];
  int * otype = (int * ) metadataArray[swID].depAddresses0[5];
  fptype * OptionPrice = (fptype * ) metadataArray[swID].depAddresses0[6];
  float timet = 0. f;
  int & bsize = BSIZE;

  int i;

  for (i = 0; i < bsize; i++) {
    fptype xStockPrice;
    fptype xStrikePrice;
    fptype xRiskFreeRate;
    fptype xVolatility;
    fptype xTime;
    fptype xSqrtTime;

    fptype logValues;
    fptype xLogTerm;
    fptype xD1;
    fptype xD2;
    fptype xPowerTerm;
    fptype xDen;
    fptype d1;
    fptype d2;
    fptype FutureValueX;
    fptype NofXd1;
    fptype NofXd2;
    fptype NegNofXd1;
    fptype NegNofXd2;

    xStockPrice = sptprice[i];
    xStrikePrice = strike[i];
    xRiskFreeRate = rate[i];
    xVolatility = volatility[i];

    xTime = time[i];
    xSqrtTime = sqrt(xTime);

    logValues = log(sptprice[i] / strike[i]);

    xLogTerm = logValues;

    xPowerTerm = xVolatility * xVolatility;
    xPowerTerm = xPowerTerm * 0.5;

    xD1 = xRiskFreeRate + xPowerTerm;
    xD1 = xD1 * xTime;
    xD1 = xD1 + xLogTerm;

    xDen = xVolatility * xSqrtTime;
    xD1 = xD1 / xDen;
    xD2 = xD1 - xDen;

    d1 = xD1;
    d2 = xD2;

    NofXd1 = CNDF(d1);
    NofXd2 = CNDF(d2);

    FutureValueX = strike[i] * (exp(-(rate[i]) * (time[i])));
    if (otype[i] == 0) {
      OptionPrice[i] = (sptprice[i] * NofXd1) - (FutureValueX * NofXd2);
    } else {
      NegNofXd1 = (1.0 - NofXd1);
      NegNofXd2 = (1.0 - NofXd2);
      OptionPrice[i] = (FutureValueX * NegNofXd2) - (sptprice[i] * NegNofXd1);
    }
  }
}
```

Listing 2: *Code of the Blackscholes function that is wrapped by tasks. This function was modified to (1) take only the* `SWID` *parameter and (2) retrieve its original parameters by querying the* `metadataArray`.

```
void BlkSchlsEqEuroNoDiv_inline(fptype * sptprice,
  fptype * strike, fptype * rate, fptype * volatility,
  fptype * time, int * otype, float timet, fptype * OptionPrice, int size) {
  int i;

  for (i = 0; i < size; i++) {
    fptype xStockPrice;
    fptype xStrikePrice;
    fptype xRiskFreeRate;
    fptype xVolatility;
    fptype xTime;
    fptype xSqrtTime;

    fptype logValues;
    fptype xLogTerm;
    fptype xD1;
    fptype xD2;
    fptype xPowerTerm;
    fptype xDen;
    fptype d1;
    fptype d2;
    fptype FutureValueX;
    fptype NofXd1;
    fptype NofXd2;
    fptype NegNofXd1;
    fptype NegNofXd2;

    xStockPrice = sptprice[i];
    xStrikePrice = strike[i];
    xRiskFreeRate = rate[i];
    xVolatility = volatility[i];

    xTime = time[i];
    xSqrtTime = sqrt(xTime);

    logValues = log(sptprice[i] / strike[i]);

    xLogTerm = logValues;

    xPowerTerm = xVolatility * xVolatility;
    xPowerTerm = xPowerTerm * 0.5;

    xD1 = xRiskFreeRate + xPowerTerm;
    xD1 = xD1 * xTime;
    xD1 = xD1 + xLogTerm;

    xDen = xVolatility * xSqrtTime;
    xD1 = xD1 / xDen;
    xD2 = xD1 - xDen;

    d1 = xD1;
    d2 = xD2;

    NofXd1 = CNDF(d1);
    NofXd2 = CNDF(d2);

    FutureValueX = strike[i] * (exp(-(rate[i]) * (time[i])));
    if (otype[i] == 0) {
      OptionPrice[i] = (sptprice[i] * NofXd1) - (FutureValueX * NofXd2);
    } else {
      NegNofXd1 = (1.0 - NofXd1);
      NegNofXd2 = (1.0 - NofXd2);
      OptionPrice[i] = (FutureValueX * NegNofXd2) - (sptprice[i] * NegNofXd1);
    }
  }
}
```

Listing 3: *Original, unmodified version of the Blackscholes function depicted in Listing 2.*

5.2.3 ORD-Phentos API

```
inline void femtos_init()
```

High-level description: initialization function that should be called before any other ORD-Phentos method is executed.

Implementation details: spawns worker threads, binds them to corresponding cores, and writes initial values to Phentos internal variables.

```
inline unsigned getNewSWID (const unsigned firstIDNotToBeUsed)
```

High-level description: finds the first available entry in the metadata array and returns the index of such entry.

Implementation details: this function checks whether a certain position of the metadata array is available by simply checking whether the function pointer it encapsulates is equal to zero; ORD-Phentos makes sure that once a task is retired, its corresponding function pointer in this metadata array is set to zero.

```
inline unsigned long long make_submission_request(int numPackets, int coreID)
```

High-level description: attempts to perform a submission request for a task to be described by `numPackets` packets; it returns -1 in case of failure; `coreID` identifies the core from which that task will be submitted.

Implementation details: this function is merely a wrapper to the *Submission Request* instruction; it does not have any side-effects on Phentos internal variables.

```
inline unsigned long long submit_packet(unsigned long long packet, int
↪ coreID)
```

High-level description: attempts to submit a single submission packet described by `packet`; the `coreID` parameter currently does not have any effect.

Implementation details: this function is merely a wrapper to the *Submit Packet* instruction; it does not have any side-effects on Phentos internal variables.

```
inline unsigned long long submit_three_packets(unsigned long long bigPacket,
↪ unsigned long long smallPacket)
```

High-level description: attempts to submit three ordered submission packets `(P1, P2, P3)` at once; `bigPacket` should be set as `((P1 << 32) | P2)` — the concatenation of `P1` and `P2` —, while `smallPacket` should equal `P3`; either all three packets are successfully submitted or none are; in case of failure, it returns -1.

Implementation details: this function is merely a wrapper to the *Submit Three Packets* instruction; it does not have any side-effects on Phentos internal variables.

```
#define make_submission_request_or_work(numPackets, coreID,
↪ numPendingWorkRequests)
```

High-level description: continuously attempts to make a submission request for a task to be described by `numPackets` packets; `coreID` identifies the core from which that task will be submitted; only returns once the submission is successful; after every unsuccessful submission attempt, it tries to fetch and execute a ready task; `numPendingWorkRequests` is a pass-by-reference parameter that should contain the number of pending work requests for the core in which this method in going to be run; if `numPendingWorkRequests` is correctly set at call time, it makes sure that it reflects the updated number of pending work requests once the method returns.

Implementation details: it uses the *Submission Request* instruction for attempting submission requests; every time a request fails fails, it attempts to fetch and execute a new ready task; once it has succeeded or failed at fetching and executing a task, it tries to perform the submission request again.

```
#define submit_or_work(packet, coreID, numPendingWorkRequests)
```

High-level description: continuously attempts to submit a single submission packet described by `packet`; the `numPendingWorkRequests` should be set to the number of work-fetch requests that have so far been made and that are not yet fulfilled; the `coreID` parameter currently does not have any effect; only returns once the submission is successful; after every unsuccessful submission attempt, it tries to fetch and execute a ready task.

Implementation details: it uses the *Submit Packet* instruction for attempting a single-packet submission; every time a submission fails, it attempts to fetch and execute a new ready task; once it has succeeded or failed at fetching and executing a task, it tries to perform the packet submission again.

```
#define submit_three_or_work(bigPacket, smallPacket, numPendingWorkRequests)
```

High-level description: continuously attempts to submit three ordered submission packets `(P1, P2, P3)` at once; `bigPacket` should be set as `((P1 << 32) | P2)` — the concatenation of `P1` and `P2` —, while `smallPacket` should equal `P3`; either all three packets are successfully submitted or none are; `numPendingWorkRequests` is a pass-by-reference parameter that should contain the number of pending work requests for the core in which this method in going to be run; if `numPendingWorkRequests` is correctly set at call time, it makes sure that it reflects the updated number of pending work requests once the method returns; only returns once the submission is successful; after every unsuccessful submission attempt, it tries to fetch and execute a ready task.

Implementation details: it uses the *Submit Three Packets* instruction for attempting three-packet submissions; every time a submission fails, it attempts to fetch and execute a new ready task; once it has succeeded or failed at fetching and executing a task, it tries to perform the three-packet submission again.

```
inline void task_wait_and_try_executing_tasks (unsigned long long
↪  numberOfSubmittedTasks)
```

High-level description: function that blocks until the total number of tasks retired by all cores equals `numberOfSubmittedTasks`; while blocked, it continuously attempts to fetch and execute ready tasks.

Implementation details: checks if `numberOfSubmittedTasks` equals `numRetiredTasks` repeatedly; whenever that condition is verified to be false, it attempts to fetch and execute a ready task; in case it succeeds at running a task, it updates a function-private variable tracking the number of tasks run and retired by the current call of this function; if it fails to run a task, it may update `numRetiredTasks` and reset its function-private retirement counter; to reduce cache bouncing related to querying and updating `numRetiredTasks`, it reduces the frequency of access to that variable by only allowing it to be updated when the function-private retirement counter is non-zero and the number of attempts at fetching and running tasks since this function was called is a multiple of a certain fixed number.

```
inline void task_wait (unsigned numberOfSubmittedTasks)
```

High-level description: function that blocks until the total number of tasks retired by all cores equals `numberOfSubmittedTasks`

Implementation details: this function will busy-wait until the `numRetiredTasks` counter equals `numberOfSubmittedTasks`; for avoiding excessive cache bouncing related to pooling the retirement counter, the interval between two subsequent pooling actions is guaranteed to exceed a fixed number of cycles.

```
inline void report_execution_stats()
```

High-level description: if the `ENABLE_PROFILING` macro is defined, it prints a report describing the number of executed tasks since system initialization, the total number of cycles of all executed tasks, and mean task size measured in cycles; if `ENABLE_PROFILING` is not defined, it prints the message *"Profiling was not enabled during Phentos compilation"* and returns.

Implementation details: it trivially calculates the average task size from the number of tasks and the total execution time of all tasks spawned since Phentos initialization.

```
#define ENABLE_PROFILING
```

High-level description: defining this macro enables the `report_execution_stats()` function to produce its performance report; if it is not defined, that function will produce a failure message and return.

Implementation details: defining this macro makes sure that the execution time of every task executed by the system is measured and added to `totalExecutedTaskCycles`; execution time of each task is measured using the `rdcycle` RISC-V instruction; the reporting function `report_execution_stats()` uses such information for calculating the average size of the tasks spawned since the initialization of Phentos.

```
#define METADATA_ARRAY_SIZE 512

struct task_metadata {
        uint64_t functionAddr;
        uint64_t depAddresses0[7];
#ifdef EXTRA_DEPS
        uint64_t depAddresses1[8];
#endif
} __attribute__ ((aligned (64)));

TaskMetadata __attribute__((aligned(64))) metadataArray[METADATA_ARRAY_SIZE];
```

Listing 4: *Definition of `metadataArray` and related data structures.*

High-level description: once a ready task is fetched and its encapsulated `SWID` is read, ORD-Phentos should find information about the pointers of its functions and data dependencies; such information should be written by the Phentos-based application into this `metadataArray` just before the task is submitted; if the `EXTRA_DEPS` macro is defined, each task might have up to 15 data dependencies (the maximum supported by Picos); if `EXTRA_DEPS` is not defined, the number of data dependencies per task is limited to 7.

```
#define EXTRA_DEPS
```

High-level description: if this macro is defined, each element of the `metadataArray` might hold up to 15 64-bit pointers describing data dependencies of a task, apart from the 64-bit variable for containing a relevant function pointer.

Implementation details: as depicted in Listing 4, whenever this macro is defined, each element of the `metadataArray` will include both a `uint64_t depAddresses0[7]` and a `uint64_t depAddresses1[8]` element, while it should not include the latter if this macro variable is not defined.

5.2.4 FAST-Phentos execution model

The FAST-Phentos API was developed with the aim of improving upon the performance of the ORD-Phentos API by reducing memory accesses related to management of task metadata during submission and ready-task fetching. Consequently, applications parallelized with FAST-Phentos should frequently outperform their ORD-Phentos versions. Even so, some applications might be more amenable to ORD-Phentos than to FAST-Phentos, as the following discussion should

make clear. In the remaining of this Subsection, we describe how Phentos handles task submission, fetching, execution, and retirement when Fast-Phentos is used, highlighting differences from ORD-Phentos.

Phentos initialization Once Phentos is initialized, worker threads are created and Phentos internal variables are set to default values. Worker threads will continuously try fetching and executing ready tasks until the program finishes. Phentos initialization is exemplified by line 5 of Listing 5.

Submission While ORD-Phentos keeps information about in-flight tasks in the `metadataArray`, FAST-Phentos implements a way of keeping information of tasks that completely avoids writes to such data structure, improving submission performance. Instead of taking up to 1024 bits for describing a task as ORD-Phentos, FAST-Phentos requires the metadata of each task to be compressed to a single 64-bit word, which should be passed to Picos as the task `SWID` and retrieved back from it during work-fetching.

The use of FAST-Phentos depends on the application ability to compress the metadata of every one of its tasks to that 64-bit limit. It is also required that the application itself implement the decompression of data dependencies. If that is not viable for a certain application, then it cannot be parallelized with FAST-Phentos and should use ORD-Phentos instead.

On the other hand, FAST-Phentos eases the compression of function pointers by providing the `functionAddresses` array. Having the pointers of all the N functions that might be run as tasks in that array allows the application to use simple indexes taking $B(N) = \lceil log2(N) \rceil$ bits instead of full 64-bit pointers for describing the functions encapsulated by tasks. If $N = 8$, this strategy alone allows reducing task metadata by 61 bits. The use of `functionAddresses` is exemplified by lines 15 to 18 of Listing 5.

It is frequent that the data dependencies of a task refer to pointers to arrays or matrices. Such pointers might be calculated as the sum between a base address and an offset. That offset O will then be such that $O = z * E_{size}$, where E_{size} is the size of a single element of the array or matrix and z is a natural number. Based on this observation, one might then compress the data dependencies of a task by expressing them solely by their z factor, which will frequently be able to be represented with no more than 10 bits. This compression strategy is exemplified by lines (28, 38, 49, 66) of Listing 5.

In order to reconstruct each of the 64-bit data dependencies, functions will then just need information about the base addresses corresponding to each of its parameters. In some cases, those base addresses will be fixed once the arrays and matrices are allocated. In other cases, they might be subject to change. Regardless, the application might make the base address information available to functions through global variables. The reconstruction of function parameters by FAST-Phentos tasks is exemplified by lines 4 to 10 of Listing 6.

Once the application has fit all the data dependencies into a corresponding 64-bit `SWID`, it might proceed to submit the task as usual, issuing a submission request followed by a sequence of packet submission operations. This is exemplified by lines (31–33, 41–44, 53–55, 70–72) of Listing 5.

It should be noted that even though the `SWID` contains all the information needed by the application to reassemble the characteristics of a task once it comes back from Picos as a ready task, Picos still needs to receive all task metadata in its ordinary uncompressed format (see Figure 4.4). Consequently, in comparison with ORD-Phentos, FAST-Phentos does not reduce the number of packets that should be submitted to Picos: it just reduces the number of memory operations needed for storing and retrieving task metadata to and from shared memory, possibly leading to sensible performance gains.

Work-fetching and Execution In a system with N cores, Phentos will initialize $N-1$ worker threads during its initialization. All such threads continuously try fetching and executing ready tasks. Additionally, several blocking functions that might be called by the submission thread allow tasks to be fetched and executed while the unblocking condition is not reached.

The functions encapsulated by FAST-Phentos tasks should expect their `SWID` as their only parameter. While ORD-Phentos functions would use that `SWID` to retrieve information about their data dependencies from the `metadataArray`, FAST-Phentos reconstructs information about its data dependencies by decompressing the `SWID` itself with possibly the help of shared variables describing the base addresses of its pointer parameters.

Once the data dependencies of a function are reconstructed, it executes its payload as usual. If profiling is enabled by the `ENABLE_PROFILING` macro, FAST-Phentos will evaluate the execution time of every task and add such value to the `totalExecutedTaskCycles` atomic variable.

Retirement Once a task is retired, FAST-Phentos informs Picos that the task has finished executing — possibly making more tasks ready —, and increments its atomic `numRetiredTasks` counter.

Task wait As described in Section 2.6, task waiting is the action by which a program blocks until a certain number of tasks have been retired. By keeping track of the number of all tasks that have been retired by any thread using the `numRetiredTasks` shared counter, Phentos allows this functionality to be implemented by simply busy-waiting until that counter has reached some given number of tasks. This is the behavior of the `task_wait()` function implemented by both ORD-Phentos and FAST-Phentos.

Instead of just waiting for the task wait condition to be reached, though, both ORD-Phentos and FAST-Phentos provide alternative task waiting functions that execute tasks while waiting for the target number of task retirements to be reached. FAST-Phentos offer this functionality through its `fast_task_wait_and_try_executing_tasks()` function.

```
void sparselu_par_call(float ** BENCH, int matrix_size, int submatrix_size) {
  static bool femtos_initialized = false;

  if (!femtos_initialized) {
    femtos_fast_init();
    femtos_initialized = true;
  }
  unsigned num_iterations = 0;
  unsigned numPendingWorkRequests = 0;

  GLOBAL_BENCH = BENCH;
  GLOBAL_matrix_size = matrix_size;
  GLOBAL_submatrix_size = submatrix_size;

  functionAddresses[0] = (unsigned long long) FA_lu0;
  functionAddresses[1] = (unsigned long long) FA_fwd;
  functionAddresses[2] = (unsigned long long) FA_bdiv;
  functionAddresses[3] = (unsigned long long) FA_bmod;

  resetTaskWaitCounters();

  asm volatile("fence"::: "memory");

  unsigned long long ii, jj, kk;
  unsigned long long swID; {
    for (kk = 0; kk < matrix_size; kk++) {

      swID = ((kk & MASK15BITS) << 4) | 0x1;

      num_iterations++;
      make_submission_request_or_work_fast(6, 0, numPendingWorkRequests);
      submit_three_or_work_fast(swID, 1, numPendingWorkRequests);
      submit_three_or_work_fast((unsigned long long) BENCH[kk * matrix_size + kk], 2, numPendingWorkRequests);

      for (jj = kk + 1; jj < matrix_size; jj++)
        if (BENCH[kk * matrix_size + jj] != NULL) {

          swID = ((((kk & MASK15BITS) << 15) | (jj & MASK15BITS)) << 4) | 0x3;

          num_iterations++;
          make_submission_request_or_work_fast(9, 0, numPendingWorkRequests);
          submit_three_or_work_fast(swID, 2, numPendingWorkRequests);
          submit_three_or_work_fast((unsigned long long) BENCH[kk * matrix_size + kk], 0, numPendingWorkRequests);
          submit_three_or_work_fast((unsigned long long) BENCH[kk * matrix_size + jj], 2, numPendingWorkRequests);
        }

      for (ii = kk + 1; ii < matrix_size; ii++)
        if (BENCH[ii * matrix_size + kk] != NULL) {
          swID = ((((kk & MASK15BITS) << 15) | (ii & MASK15BITS)) << 4) | 0x5;

          num_iterations++;
          make_submission_request_or_work_fast(9, 0, numPendingWorkRequests);
          submit_three_or_work_fast(swID, 2, numPendingWorkRequests);
          submit_three_or_work_fast((unsigned long long) BENCH[kk * matrix_size + kk], 0, numPendingWorkRequests);
          submit_three_or_work_fast((unsigned long long) BENCH[ii * matrix_size + kk], 2, numPendingWorkRequests);
        }

      for (ii = kk + 1; ii < matrix_size; ii++)
        if (BENCH[ii * matrix_size + kk] != NULL) {
          for (jj = kk + 1; jj < matrix_size; jj++)
            if (BENCH[kk * matrix_size + jj] != NULL) {
              if (BENCH[ii * matrix_size + jj] == NULL) {
                BENCH[ii * matrix_size + jj] = allocate_clean_block(submatrix_size);
              }

              swID = ((((kk & MASK15BITS) << 30) | ((ii & MASK15BITS) << 15) | (jj & MASK15BITS)) << 4) | 0x7;

              num_iterations++;
              make_submission_request_or_work_fast(12, 0, numPendingWorkRequests);
              submit_three_or_work_fast(swID, 3, numPendingWorkRequests);
              submit_three_or_work_fast((unsigned long long) BENCH[ii * matrix_size + kk], 0, numPendingWorkRequests);
              submit_three_or_work_fast((unsigned long long) BENCH[kk * matrix_size + jj], 0, numPendingWorkRequests);

              asm volatile("fence"::: "memory");

              submit_three_or_work_fast((unsigned long long) BENCH[ii * matrix_size + jj], 2, numPendingWorkRequests);
            }
        }
    }
  }

  printf("Going to task wait until %d tasks were retired.\n", num_iterations);
  fast_task_wait_and_try_executing_tasks(num_iterations);
}
```

Listing 5: *Task-spawning code from the SparseLU benchmark, parallelized with the FAST-Phentos API.*

```
void FA_bmod(unsigned long long swID) {
  int i, j, k;

  int kk = (swID >> 34) & MASK15BITS;
  int ii = (swID >> 19) & MASK15BITS;
  int jj = (swID >> 4) & MASK15BITS;
  float * row = GLOBAL_BENCH[ii * GLOBAL_matrix_size + kk];
  float * col = GLOBAL_BENCH[kk * GLOBAL_matrix_size + jj];
  float * inner = GLOBAL_BENCH[ii * GLOBAL_matrix_size + jj];
  int & submatrix_size = GLOBAL_submatrix_size;

  for (i = 0; i < submatrix_size; i++)
    for (k = 0; k < submatrix_size; k++)
      for (j = 0; j < submatrix_size; j++)
        inner[i * submatrix_size + j] = inner[i * submatrix_size + j] - row[i * submatrix_size + k] * col[k * submatrix_size +
        ↪  j];
}
```

Listing 6: *Code of one of the SparseLU functions that are wrapped by tasks. This function was modified to (1) take only the* `SWID` *parameter and (2) reconstruct its remaining original parameters from information compressed into* `SWID` *and from global variables describing relevant base pointers and matrix sizes.*

```
void bmod(float * row, float * col, float * inner, int submatrix_size) {
  int i, j, k;
  for (i = 0; i < submatrix_size; i++)
    for (k = 0; k < submatrix_size; k++)
      for (j = 0; j < submatrix_size; j++)
        inner[i * submatrix_size + j] = inner[i * submatrix_size + j] - row[i * submatrix_size + k] * col[k * submatrix_size +
        ↪  j];
}
```

Listing 7: *Original, unmodified version of the SparseLU function depicted in Listing 6.*

5.2.5 FAST-Phentos API

```
inline void femtos_fast_init()
```

High-level description: initialization function that should be called before any other FAST-Phentos method is executed.

Implementation details: spawns worker threads, binds them to corresponding cores, and writes initial values to Phentos internal variables.

```
inline unsigned long long make_submission_request(int numPackets, int coreID)
```

High-level description: attempts to perform a submission request for a task to be described by `numPackets` packets; it returns -1 in case of failure; `coreID` identifies the core from which that task will be submitted.

Implementation details: this function is merely a wrapper to the *Submission Request* instruction; it does not have any side-effects on Phentos internal variables.

```
inline unsigned long long submit_three_packets(unsigned long long bigPacket,
↪  unsigned long long smallPacket)
```

High-level description: attempts to submit three ordered submission packets `(P1, P2, P3)` at once; `bigPacket` should be set as `((P1 << 32) | P2)` — the concatenation of `P1` and `P2` —, while `smallPacket` should equal `P3`; either all three packets are successfully submitted or none are; in case of failure, it returns -1.

Implementation details: this function is merely a wrapper to the *Submit Three Packets* instruction; it does not have any side-effects on Phentos internal variables.

```
#define make_submission_request_or_work_fast(numPackets, coreID,
↪  numPendingWorkRequests)
```

High-level description: continuously attempts to make a submission request for a task to be described by `numPackets` packets; `coreID` identifies the core from which that task will be submitted; only returns once the submission is successful; after every unsuccessful submission attempt, it tries to fetch and execute a ready task; `numPendingWorkRequests` is a pass-by-reference parameter that should contain the number of pending work requests for the core in which this method in going to be run; if `numPendingWorkRequests` is correctly set at call time, it makes sure that it reflects the updated number of pending work requests once the method returns.

Implementation details: it uses the *Submission Request* instruction for attempting submission requests; every time a request fails fails, it attempts to fetch and execute a new ready task; once it has succeeded or failed at fetching and executing a task, it tries to perform the submission request again.

```
#define submit_three_or_work_fast(bigPacket, smallPacket,
↪  numPendingWorkRequests)
```

High-level description: continuously attempts to submit three ordered submission packets `(P1, P2, P3)` at once; `bigPacket` should be set as `((P1 << 32) | P2)` — the concatenation of `P1` and `P2` —, while `smallPacket` should equal `P3`; either all three packets are successfully submitted or none are; `numPendingWorkRequests` is a pass-by-reference parameter that should contain the number of pending work requests for the core in which this method in going to be run; if `numPendingWorkRequests` is correctly set at call time, it makes sure that it reflects the updated number of pending work requests once the method returns; only returns once the submission is successful; after every unsuccessful submission attempt, it tries to fetch and execute a ready task.

Implementation details: it uses the *Submit Three Packets* instruction for attempting three-packet submissions; every time a submission fails, it attempts to fetch and execute a new ready task; once it has succeeded or failed at fetching and executing a task, it tries to perform the three-packet submission again.

```
inline void task_wait (unsigned numberOfSubmittedTasks)
```

High-level description: function that blocks until the total number of tasks retired by all cores equals `numberOfSubmittedTasks`

Implementation details: this function will busy-wait until the `numRetiredTasks` counter equals `numberOfSubmittedTasks`; for avoiding excessive cache bouncing related to pooling the retirement counter, the interval between two subsequent pooling actions is guaranteed to exceed a fixed number of cycles.

```
inline void fast_task_wait_and_try_executing_tasks (unsigned long long
↪  numberOfSubmittedTasks)
```

High-level description: function that blocks until the total number of tasks retired by all cores equals `numberOfSubmittedTasks`; while blocked, it continuously attempts to fetch and execute ready tasks.

Implementation details: checks if `numberOfSubmittedTasks` equals `numRetiredTasks` repeatedly; whenever that condition is verified to be false, it attempts to fetch and execute a ready task; in case it succeeds at running a task, it updates a function-private variable tracking the number of tasks run and retired by the current call of this function; if it fails to run a task, it may update `numRetiredTasks` and reset its function-private retirement counter; to reduce cache bouncing related to querying and updating `numRetiredTasks`, it reduces the frequency of access to that variable by only allowing it to be updated when the function-private retirement counter is non-zero and the number of attempts at fetching and running tasks since this function was called is a multiple of a certain fixed number.

```
inline void report_execution_stats()
```

High-level description: if the `ENABLE_PROFILING` macro is defined, it prints a report describing the number of executed tasks since system initialization, the total number of cycles of all executed tasks, and mean task size measured in cycles; if `ENABLE_PROFILING` is not defined, it prints the message *"Profiling was not enabled during Phentos compilation"* and returns.

Implementation details: it trivially calculates the average task size from the number of tasks and the total execution time of all tasks spawned since Phentos initialization.

```
uint64_t functionAddresses[NUM_DIFFERENT_TASKS]
```

High-level description: before any task is submitted, the Phentos-based application should populate this array with pointers to all functions that it might spawn as tasks; the size of the array is `NUM_DIFFERENT_TASKS`, so it might be populated from its element 0 up to its element `NUM_DIFFERENT_TASKS` − 1.

Implementation details: static array capable of holding `NUM_DIFFERENT_TASKS` elements; it allows FAST-Phentos methods to use small indexes in the [0, `NUM_DIFFERENT_TASKS`[range instead of full 64-bit pointers for describing the functions encapsulated by a task, which is beneficial for minimizing the memory footprint of task metadata structures, as discussed in Subsection 5.2.4.

```
#define NUM_DIFFERENT_TASKS 8
```

High-level description: defines the maximum number of different functions that might be encapsulated by tasks by the application compiled with Phentos.

Implementation details: this definition is needed for allocating the `functionAddresses` array at compile time.

```
#define ENABLE_PROFILING
```

High-level description: defining this macro enables the `report_execution_stats()` function to produce its performance report; if it is not defined, that function will produce a failure message and return.

Implementation details: defining this macro makes sure that the execution time of every task executed by the system is measured and added to `totalExecutedTaskCycles`; execution time of each task is measured using the `rdcycle` RISC-V instruction; the reporting function `report_execution_stats()` uses such information for calculating the average size of the tasks spawned since the initialization of Phentos.

Chapter 6

Experimental Evaluation

In this chapter we evaluate the following hypotheses:

1. that Phentos usually leads to better performance than Nanos-RV or Nanos-SW;
2. that Nanos-RV usually leads to better performance than Nanos-SW;
3. that the performance gap between the three runtimes decreases as task granularity increases;
4. that the optimization strategies implicit in the design goals of Phentos described in 5.2 are generally conducive to better performance.

Verifying the two first hypotheses implies that the architecture described in Section 4 succeeds at accelerating Task Scheduling workloads, while the third hypothesis implies that the performance gains offered by this architecture are more significant for fine-grained workloads. Finally, evaluating the fourth hypothesis should be useful for developing higher-performance or simpler versions of Phentos in the future, given that it could identify optimizations that in spite of increasing code complexity do not lead to substantially better execution times.

6.1 Methodology

The performance of the different Task Scheduling runtimes — making use of HW assistance or not — is evaluated by running the inputs of varying task granularities of the benchmarks described in Sub-section 6.1.2. The same experiments are also useful for assessing the relationship between task granularity and the performance gap between the different runtimes. Processor parameters that are influential on benchmark performance are described in Sub-section 6.1.1.

6.1.1 System parameters

The Rocket Chip prototype used in this work is an in-order eight-core processor with eight-way, 32KB, core-specific, cache-coherent L1 data and instruction caches implementing the MESI protocol. A shared L2 cache is absent, meaning that data movement between them must be mediated by main memory. Consequently, the performance of this system is specially sensitive to inter-core synchronization and L1 cache misses. On the other hand, this effect is mitigated by the fact that main memory runs at a much higher clock — 667 MHz — than the modified Rocket Chip — which

runs at 80 MHz. Benchmarks are executed on a minimal SMP-capable Linux image. All benchmark versions — including the serial ones — are compiled with `-O3` optimization strength. Also, OmpSs applications are compiled with the Mercurium compiler [Balart et al., 2004], wheres benchmark versions targeting either serial or Phentos execution are compiled with plain GCC or G++. The fact that a high optimization strength is used also for the serial versions of the benchmarks is one of the main reasons why Nanos-SW, Nanos-RV, and Phentos speedups over serial execution do not exceed the 6x factor. Experiments not included for brevity in this work show that `-O0` compiled Tasks Parallel applications might have speedups over 7x over corresponding executions of `-O0` serial binaries.

6.1.2 Benchmarks

System performance is evaluated with programs from three different domains, as described next:

1. The *blackscholes* application, representing the Financial Analysis domain, solves the Black-Scholes partial differential equation for evaluating how the price of an European-style option varies as a result of changes to the value of its underlying asset. Its implementation is based on the code found in the `parsec-ompss`[1] GitLab repository, which augments the Parsec benchmark suite [Bienia and Li, 2009] by offering OmpSs task-based implementations for most of its benchmarks. It is a highly data-parallel application.

2. The *sparseLU* and the *jacobi* applications represent the Fundamental Linear Algebra domain. The first of them solves pseudo-random sparse linear systems, while the second uses the Jacobi iterative equation solver for solving the Poisson equation in one dimension. Such programs are derived from the implementations found in the Kastors Benchmark Suite [Virouleau et al., 2014].

3. The *stream-deps* and the *stream-barr* programs are micro-benchmarks that evaluate system system performance at handling memory intense computation. Examples of these routines include copying data among memory positions; adding two arrays and storing the result in a third; producing scaled versions of an original array, etc. The fact that these benchmarks compound these operations in a complex scheme of data dependencies make them good targets for parallelization using Task Scheduling. The implementations of these benchmarks that are here used are found at the `ompss-ee`[2] Github repository.

Each of these benchmarks might be executed with inputs of varying task granularity, which is frequently achieved by partitioning input matrices in blocks of arbitrary size.

6.2 Results and Analysis

6.2.1 Comparing Nanos-SW, Nanos-RV, and Phentos

Figure 6.1 compares benchmark performance for the three available runtimes and all relevant benchmark inputs. As expected, performance of Nanos-RV, which makes use of the custom Task

[1] https://pm.bsc.es/gitlab/benchmarks/parsec-ompss
[2] https://github.com/bsc-pm/ompss-ee

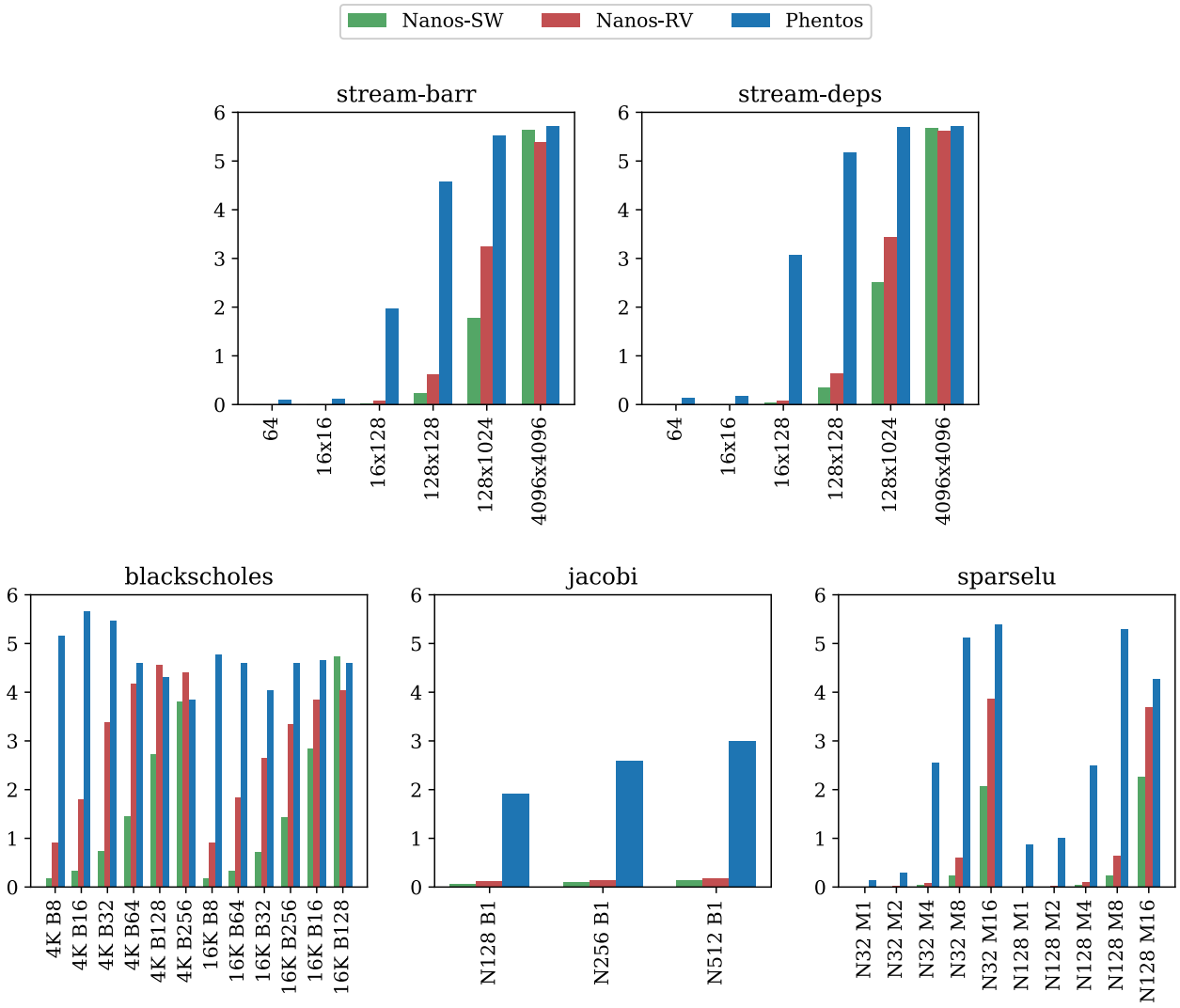

Figure 6.1: *Normalized benchmark performance for all 37 studied workloads.*

Scheduling instructions, is generally (34 out of 37 times) superior to that of Nanos-SW, which does not make use of these instructions, with a geomean speedup of 2.13 times with respect to the latter.

Additionally, the normalized performance of Phentos is almost always (36 out of 37 times) higher than that of Nanos-SW and generally (34 out of 37 times) better than that of Nanos-RV. Such gains result from the fact that it was designed to have lower synchronization and intra-core communication overheads than any version of Nanos. Its geomean speedup with respect to Nanos-RV is of 6.20 times, while its geomean speedup over Nanos-SW is of 13.19 times.

For Nanos-SW and Nanos-RV, increasing block size of the benchmarks supporting that option generally increases performance. This is less frequently true for Phentos, though. This difference between the two sets of runtimes is probably caused by the facts that (1) Phentos generates much less scheduling overheads than the other two runtimes, making the Task Scheduling overhead reductions caused by increasing block sizes less meaningful; (2) larger block sizes reduce the number of generated tasks, possibly reducing available parallelism and amplifying load balancing problems; (3) depending on how each benchmark was implemented, increasing block size might even lead to worse cache usage.

For stream-deps and stream-barr, performance increases for all runtimes as problem size is increased. For these programs, block size is defined as a fixed fraction of problem size.

6.2.2 Deriving theoretical speedup bounds from MTT

As described in Subsection 2.6, Maximum Task Throughput (MTT) is the maximum number of tasks from a uniform-task-size workload that a given Task Scheduling platform might execute per unit of time. This metric is very important for comparing different Task Scheduling systems, given that it defines constraints for the (task granularity, number of cores) pairs that such systems are able to efficiently service.

In fact, in a system with N cores being served by a Task Scheduling runtime with an MTT of K, the following inequality must hold:

$$\frac{N_{active}}{T_{exec}} \leq K,$$

where T_{exec} is the fixed task execution time measured in cycles and N_{active} is the average number of cores actively running tasks — rather than waiting to be fed with more work by the Task Scheduling runtime. In this equation, N_{active} is dimensionless, T_{exec} has the dimension of time, and K has the dimension of the inverse of time (which naturally follows from the fact that it is a kind of frequency).

In any situation, N_{active} corresponds to the maximum speedup that the task parallel version of an application might offer with respect to its serial counterpart. Having, for example, $N_{active} = 3$ would mean that, on average, at most three cores would be actually running tasks, while all others would simply be starving for work. In such situation, the speedup of the task parallel application should not be higher than 3.

Based on that logic, one might relate the speedup $S \leq N_{active}$ of the task parallel version of an application with respect to serial execution for that system as the following:

$$S(T_{exec}) \leq N_{active} \leq K \times T_{exec}$$

Additionally, it is easy to see that:

$$K \leq \frac{1}{L_o},$$

where L_o is the mean Task Scheduling overhead experienced by tasks during their whole lifetime. Such inequality might be illustrated by the fact that a Task Scheduling system should not be able to dispatch more than, for example, 1000 tasks per second if each task had a lifetime Task Scheduling overhead of 1 millisecond, given that the total overhead of all those 1000 tasks should fit in a single second.

From there it follows that:

$$S(T_{exec}) \leq K \times T_{exec} \leq \frac{T_{exec}}{L_o}$$

If we then define MS as the maximum speedup S that an application might experience for $T_{exec} = t$ and for a given L_o, the following must be true:

$$MS(L_o, t) = \frac{t}{L_o} \tag{6.1}$$

Having this in mind, for four different workloads, we measured the mean Task Scheduling overhead of Nanos-RV and Phentos, as shown in Figure 6.2. That figure also compares with the previous

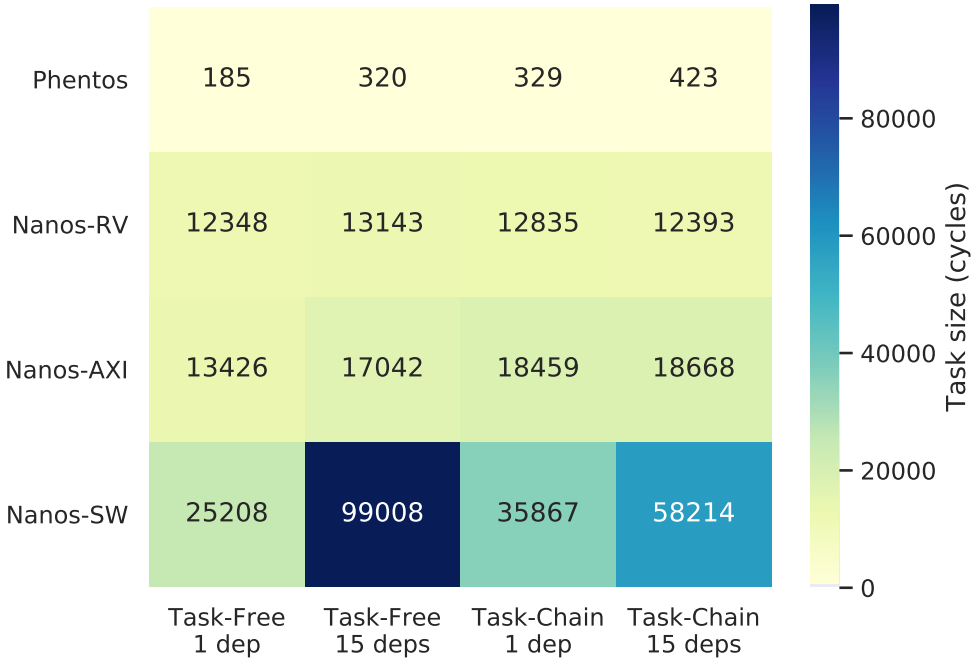

Figure 6.2: *Lifetime Task Scheduling overhead for several platforms, in Rocket Chip equivalent cycles. The Nanos-RV and Phentos platforms correspond to work described in this dissertation, while data for Nanos-AXI derives from a recent work by Tan et al. [2017]. The measurements reported by the latter work were performed on an ARM based system based on a Cortex-A9 quad-core processor, so the numbers reported in this figure are scaled by the ratio between the average instruction-per-cycle metrics of Cortex-A9 and Rocket Chip reported by Celio et al. [2015]. Consequently, the Nanos-AXI figures here reported are about 57% higher than those described by Tan et al. [2017].*

state-of-the-art Task Scheduling system based on Picos, which implemented Picos-CPU communication with asynchronous AXI transactions controlled by a dedicated DMA-like communication module [Tan et al., 2017].

Figure 6.2 clearly shows to which extent Picos-RV and Phentos were able to reduce lifetime Task Scheduling overheads for varying workloads. In fact, Phentos presents lifetime overhead reductions of up to 308x with respect to Nanos-SW, while Nanos-RV shows reductions of up to 7.53x with respect to the same baseline. Such measurements were taken with two different lifetime-overhead-measuring benchmarks: *Task Free*, which generates independent tasks with any number of dependencies from 0 to 15; and *Task Chain*, which generates inter-dependent tasks forming a data dependence chain where all tasks have the same number of dependencies similarly ranging from 0 to 15.

Applying Equation 6.1 with the data reported for *Task-Chain (1 dep)* in Figure 6.2, we might then evaluate maximum speedup bounds for the various different Task Scheduling platforms as a function of mean task size[3] as shown in Figure 6.3. That figure shows that the reduced lifetime overheads of Phentos substantially improve MTT-based maximum speedup with respect to any other platform for a wide range of mean task sizes. Concretely, for task sizes around 1000 cycles, MTT-based maximum speedup for the Phentos platform is just below 3x, while all other platforms have maximum speedups lower than 0.1x; moreover, for task sizes around 10000 cycles, the maximum speedup of Phentos has already saturated to 8x (the number of available cores), whereas all other platforms fail to deliver maximum speedups over 1x.

[3]We use the term *task size* as a synonym of *task execution time*.

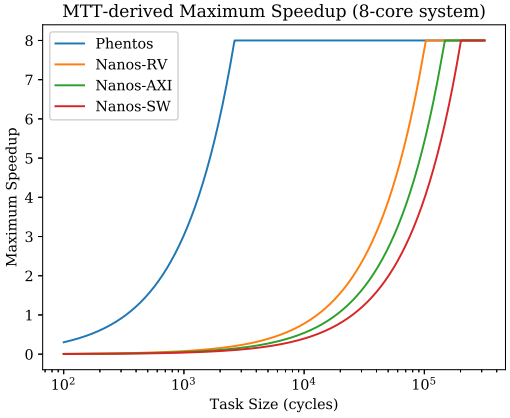

Figure 6.3: *Theoretical MTT-derived speedup bounds for several Task Scheduling platforms with eight cores.*

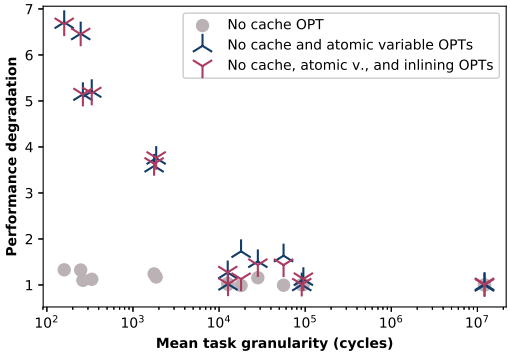

Figure 6.4: *Impact of different Phentos optimizations on benchmark performance.*

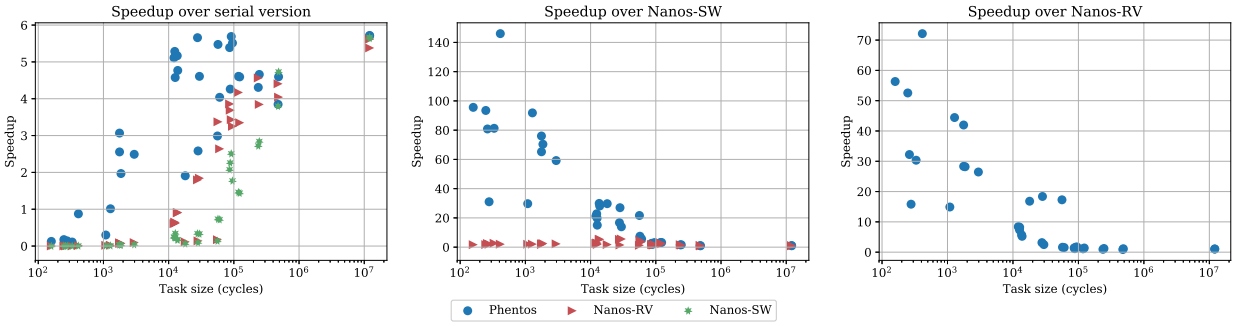

Figure 6.5: *Speedups of all 37 benchmark workloads running on each of the platforms with respect to corresponding serial executions or equivalent runs on lower-MTT platforms.*

6.2.3 Effects of task granularity

As discussed in Sub-section 6.2.2, mean task size greatly influences the maximum speedup that a given Task Scheduling system might deliver over a corresponding serial execution. Adding to that discussion, Figure 6.5 shows how the speedup of Task Scheduling platforms with respect to lower-MTT platforms and corresponding serial executions depends on mean task size. The data points there represented correspond to the same benchmark executions reported in Figure 6.1.

6.2.4 Experimental validation of Phentos optimizations

In order to better assess the impact of each of the Phentos optimizations described in Section 5.2, we evaluated how the selective activation of several of these optimizations impacts the performance of a set of three benchmarks (*Jacobi*, *stream-deps*, and *stream-barr*). Results are presented in Figure 6.4, which reports the performance degradation resulting from the disabling of three different optimizations. In this figure, one can see that disabling *cache optimizations* — which fulfills design goals (2), (3), and (4) — leads to only moderate performance degradation (1.09x on time), while disabling cache optimizations and *minimization of writes to atomic variables* — which fulfills design goal (5) — substantially increases performance degradation (to 2.15x on average); at the same time, this data shows that, in general, disabling *function inlining* does not lead to higher performance degradation, but can even lead to improved performance with respect to the system setup without

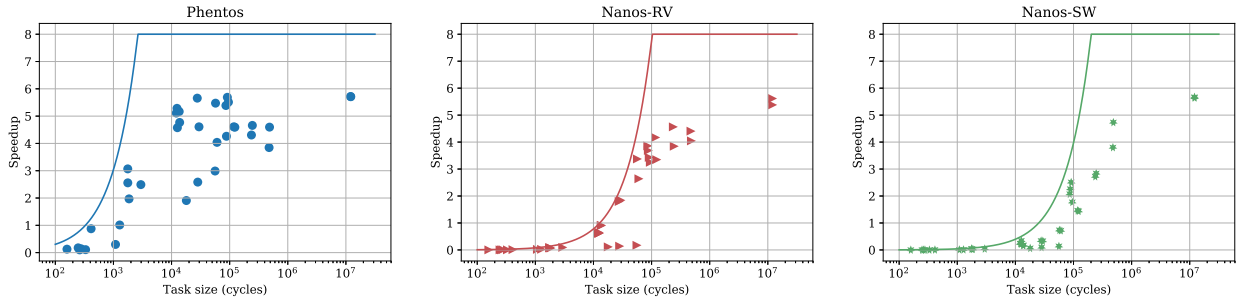

Figure 6.6: *Experimental speedup data of all 37 benchmark workloads over corresponding serial executions compared with theoretical MTT-derived bounds. As it is the case for Figure 6.3, MTT values are derived from Task-Chain executions involving one dependence per task.*

the two other optimizations for some medium-grained workloads.

For the benchmarks that are negatively impacted by function inlining, we hypothesize that the additional instruction-cache pressure caused by the expansion of inlined functions at caller sites could be offsetting the reduced calling latency of inlined functions. Such effects might be less pronounced in systems including L2 cache or larger L1 instruction caches.

In any case, considering all benchmark executions, the performance impact of the several optimizations is generally more noticeable for the most fine-grained workloads. In fact, degradation over 2x only occurs for executions involving tasks with average execution time lower than 2000 cycles.

Chapter 7

Conclusion

In this dissertation, we propose an architecture where the capabilities of a Task Scheduling accelerator are made available to Task Scheduling runtimes through (1) low-latency, custom processor instructions and (2) dedicated accelerator-CPU interconnects. Compared with previous solutions that relied on MMIO CPU-accelerator communication, our system is shown to greatly reduce lifetime scheduling overhead, leading to proportional gains in whole-application speedup. We validate the proposed architecture with a Linux-capable, 8-core FPGA prototype based on Rocket Chip, a popular SoC generator featuring a parametrizable, in-order, 64-bit, Linux-capable multi-core processor.

To evaluate performance gains of Task Parallel programs, we execute a set of OmpSs Task Parallel benchmarks on (1) Nanos-SW, an widely-available OmpSs runtime using no HW acceleration; (2) Nanos-RV, an in-house port of Nanos-SW for the new architecture; (3) Phentos, a completely new, light-weight, high-performance Task Scheduling runtime. Nanos-RV allows applications to achieve an average speedup of 2.13x with respect to executions based on Nanos-SW, while Phentos delivers an average speedup of 13.19x over the same baseline. Such gains are made possible by the fact that, making use of the low-latency Task Scheduling capabilities of the new system, lifetime scheduling overheads with respect to Nanos-SW are reduced by up to 7.53x by Nanos-RV and by up to 308.75x by Phentos.

7.1 Future Work

In this last section, we discuss some of the possible extensions to our work that we expect to be of greatest value.

7.1.1 Medium-effort extensions

- Implementing extra compiler support for ORD-Phentos and FAST-Phentos

  At its current state, Phentos gives applications ample access to its internal states and to very low-level hardware scheduling functionality. While this is one of the reasons why it can greatly outperform Nanos-RV, this is detrimental to programmer productivity.

  Fortunately, though, most of the application-Phentos interactions could be made much simpler if ampler compiler support for Phentos was provided.

- Letting Picos Manager store information about data dependencies so as to avoid the use of the `metadataArray` without the difficulties and limitations imposed by FAST-Phentos

  Doing so would allow Phentos-based applications to enjoy performance superior to that of ORD-Phentos without the limitations of FAST-Phentos, which does not support applications generating tasks whose dependencies cannot be compressed to a single 64-bit value.

- Adding support for nested tasks by replacing Picos with Picos++

  Picos and Picos++ share the same packet format. Consequently, it should be easy to replace Picos with Picos++, which has support for nested tasks. More adaptations might be needed for fully supporting tasks of that kind, but implementing them should be straightforward.

- Allowing the system to support multiple applications simultaneously

  Our system currently does not allow multiple task parallel applications to simultaneously take benefit of its Task Scheduling HW support. It should be interesting to expand our solution so as to make that possible.

7.1.2 High-effort extensions

- Creating a HW interface for allowing Picos Manager to connect not only to cores, but also to accelerators

  Picos Manager is a parametrizable design that can be easily modified to accommodate different numbers of cores. On the other hand, it does not offer any support for the management of accelerators that are independent of CPUs. It would be interesting to develop a standard Picos Interface for allowing HW accelerators to have their tasks managed by the underlying HW Task Scheduler without any interference of the CPUs.

- Implementing task-aware cache-eviction policies

  Cache-eviction policies used by modern processors might either be independent of how data is used (as dictated by Random Replacement), or implement heuristics that attempt to remove first those lines that has least recently been used (as prescribed by Pseudo LRU). It is usually expected that the latter policies outperform the former random ones, given that information about how data is being used should increase the chance that the line being evicted was not going to be used again. On the other hand, for task parallel applications, even more useful than trying to detect which cache lines have been the least recently used could be checking which cache lines refer to in-flight tasks and which ones have been orphaned since their corresponding tasks have been retired.

- Implementing data-aware task placing

  Our system currently distributes ready tasks to cores in a first-come, first-served manner. We acknowledge that this might not be the optimal strategy, as it might be frequently better to have related tasks be scheduled to the same core for minimizing inter-core data communication or better leveraging instruction cache.

- Implementing criticality-aware Task Scheduling

Apart from minimizing inter-core communication or better making use of instruction cache, the distribution of tasks could be done so as to make sure that tasks belonging to the critical path of the task graph were executed as soon as possible. In some situations, this strategy should be useful for reducing whole-program execution time.

- Implementing task-aware cache prefetching

  Ensuring that our Task Scheduling system actively plans how it should distribute tasks to cores rather than allowing that to happen randomly could also be useful for cache prefetching. If the Task Scheduler knows which tasks should start executing in a certain core after the latter finishes running a certain task, the scheduler could make cache-prefetching operations in behalf of that core to allow data of the new task to arrive in timely fashion, improving the efficiency of the cache hierarchy and consequently overall system performance.

7.2 Acknowledgements

We would like to thank our colleagues at BSC for allowing us to build upon their work on Picos and Nanos. We also thank all researchers that contributed to this work through advice or help to implement it, as well as the family members and friends that backed us along the way.

The research here depicted was conducted with the financial support of FAPESP through processes 2017/02682-2 and 2018/00687-0.

Appendix A

Development in Detail

In this Appendix, we describe the several implementation steps that had to be overcome to build the base HW-SW system on top of which Nanos-RV and Phentos were developed. In summary, those steps are the following:

- Booting SMP Linux on top of FPGA-based Rocket Chip with arbitrary numbers of cores

  Task Scheduling is only useful for multi-processed systems. Because of that, we had to make sure that we could boot SMP Linux on top of Rocket Chip instances containing more than one core. This involved finding the right combination of versions for RISC-V GCC, Linux, Rocket Chip, and other tools, as well as patching the Rocket Chip bootrom.

- Porting the OmpSs runtime library and compiler to the RISC-V architecture

  In the long term, we would like any Task Scheduling application targeting either OmpSs or OpenMP to be able to leverage the Hardware-based scheduling capabilities provided by our system. In order to realize that, one should produce OmpSs and OpenMP runtimes capable of transparently calling hardware-based Task Scheduling instructions whenever possible. That being true, we ported the current version of Nanos - the OmpSs runtime - to RISC-V as a first step of adaptation.

- Publishing a Docker image with the tools needed for building SMP Linux for Rocket Chip

  Finding the right version combination for all the tools needed for building and deploying SMP Linux on top of Rocket Chip required a large amount of time. Because of that, we thought it would be convenient to crystallize this combination of tools with an appropriate Docker image, which could be easily shared with research colleagues and later used by ourselves for replicating our early work.

- Creating and testing example RoCC accelerators

  Our addition of new instructions to Rocket Chip for letting Task Parallel programs interact with Picos was made possible by the RoCC Interface, a Rocket Chip provision for easily adding accelerators that are accessed through custom instructions. Consequently, our early hardware development process involved designing and testing simple RoCC-based accelerators for learning purposes.

- Implementing Picos Manager: Chisel Logic for integrating Picos with Rocket Chip

The hardware Task Scheduling capabilities of our final system are provided by Picos. Using Picos greatly simplified our development process, as it spared us from designing, implementing, and testing a Task Scheduler for the system. On the other hand, Picos was not designed with the goal of simultaneously serving several cores in mind. Consequently, we had to design an IP block for abstracting the existence of several cores from Picos. It achieves so by arbitrating requests coming from cores and automatically routing replies from Picos to the adequate cores. It also provides cores with a communication scheme that is simpler than that of Picos.

- Integrating Picos to Rocket Chip and running binaries on top of the end system

  The goal of implementing Picos Manager was to build an adequate interface between Picos and the rest of the multi-core system. Consequently, the integration of Picos to the system naturally followed the implementation of Picos Manager.

  Moreover, once bare-metal tests were successful, we had to ensure that benchmarks could also be invoked from within the RISC-V Linux environment. Doing so involved checking how to prevent Rocket Chip to issue "illegal instruction" traps for RoCC instructions called from within Linux.

- Porting System to Ultrascale+ MPSoC

  All of our initial development was based on the ZC706 board, from the Zynq-7000 family. This board was very appropriate for our first experiments with Rocket Chip, given that the `fpga-zynq` repository provides all tools for implementing Rocket Chip on the ZC706 FPGA. On the other hand, once we came back from our internship at BSC, we did not have access to any ZC706 board anymore, so we had to re-target our work to the Ultrascale+ MPSoC ZCU102-ES2. This involved changing the memory map of the system to accommodate differences between the two target boards as well as porting some ARM tools to the 64-bit architecture implemented by the latter board.

- Building and Booting the Final System

In the following sections, we describe each of these steps in detail.

A.1 Booting SMP Linux on FPGA-based Multi-core Rocket Chip

Our project involved using multi-core Rocket Chip as the base system for a tightly integrated Task Scheduler. That being the case, one of the first goals of the project was making sure that we would be able to boot SMP Linux on top of a Rocket Chip instance running on an FPGA. We attacked this problem in two steps, which can be summarized as follows:

1. Running Linux on top of a single-core Rocket Chip

   (a) Identifying and studying relevant Github repositories.

   (b) Leveraging pre-compiled FPGA bitstreams and Linux images for booting Linux on top of a single-core Rocket Chip running on a ZC706 Xilinx board.

2. Running SMP Linux on top of a multi-core Rocket Chip

(a) Rebuilding the Rocket Chip bitstream targeting the ZC706 board and the corresponding Linux image.

(b) Rebuilding Linux for the Rocket Chip version being tested.

(c) Identifying the right version combination for the (Rocket Chip, RISC-V GNU Toolchain, Berkeley Bootloader, Front-end Server, Linux) tuple in order to build a working Linux system with support for SMP.

(d) Generating SMP Linux images with the default Buildroot configuration included in the standard `freedom-u-sdk` project.

(e) Testing RISC-V Linux images on QEMU and Spike simulators targeting various versions of the RISC-V Privileged ISA Specification.

(f) Adapting Rocket Chip's RISC-V assembly BOOTROM code so as to let it include a binary device tree compiled from the ASCII device tree automatically generated by Rocket Chip during elaboration of the Verilog for the multi-core chip.

(g) Testing the final combination of Rocket Chip bitstream and Linux image on FPGA.

Running Linux on top of a single-core Rocket Chip can be achieved by simply using the pre-built files available at the `fpga-zynq` Github project, which provides support for the deployment of Linux-capable instances of Rocket Chip to Zynq-7000 boards. Running SMP Linux on top of a multi-core Rocket Chip, on the other hand, is only possible after performing some modifications to Rocket Chip, as we show in the rest of this Section.

A.1.1 Using a SMP-enabled bootrom

The default bootrom provided by the `master` branch of `fpga-zynq` project does not support SMP booting. On the other hand, one might find a SMP-capable bootroms in either the `sifive/freedom`[1] or the `new-devices` branch of `fpga-zynq` itself. Given the simpler nature of the latter bootrom, this was the one we used.

A bootrom is a short program that is the first code segment run by a processor as it boots. It might be used, for example, for resetting registers, setting up interrupt vectors, and jumping to a higher-level bootloader. The bootrom from `new-devices` lets a single master core proceed to the next-level bootloader while all others spin-wait for an interrupt coming from the master core. The bootrom defines that the next-level bootloader — which is the `bbl` instance containing the Linux image — should be located at the hardcoded memory position `0x80000000`.

RISC-V Linux expects to find a pointer to the system device tree in the `a1` register during early boot. In our system, the device tree (in flattened binary form) is appended to the bootrom binary itself, and its address is loaded to that register before `bbl` is called. Once it starts executing, `bbl` makes sure that this value of `a1` is preserved up to the point that Linux is called, so that Linux may in fact find a pointer to the device tree contained by the bootrom.

The full contents of the used bootrom can be found in Listing 8.

[1] https://github.com/sifive/freedom

```
#define DRAM_BASE 0x80000000

.section .text.start, "ax", @progbits
.globl _start
_start:
  li a1, 0x2000000          // pseudo-instruction for loading arbitrary value 0x2000000 to gp register a1
  csrr a0, mhartid          // mhartid is a read-only register that contains the integer ID of the HW thread now running
                            // csrr is an instruction for reading from a CSR into a GP reg
  bnez a0, boot_core        // if a0 != 0, then go to boot_core

  addi a2, a1, 4            // a2 = a1 + 4
  li a3, 1                  // a3 = 1

interrupt_loop:
  sw a3, 0(a2)              // *(a2 + 0) = a3
  addi a2, a2, 4            // a2 = a2 + 4
  lw a3, -4(a2)             // a3 = *(a2 - 4)
  bnez a3, interrupt_loop   // if a3 != 0, then go to interrupt_loop
  j boot_core               // go to boot_core

.section .text.hang, "ax", @progbits
.globl _hang
_hang:
  // This boot ROM doesn't know about any boot devices, so it just spins,
  // waiting for the serial interface to load the program and interrupt it
  la a0, _start             // a0 = PC + _start
  csrw mtvec, a0            // mtvec = a0
  li a0, 8                  // a0 = 8 (b'1000)                                  // MIE or MSIP bit
  csrw mie, a0              // mie = a0 (b'1000)                                // set only MSIP in mie CSR
  csrw mideleg, zero        // mideleg = 0                                      // no delegation
  csrs mstatus, a0          // mstatus = mstatus | a0 = mstatus | b'1000        // set MIE in mstatus CSR

wfi_loop:
  wfi                       // Wait for interrupt. It may be implemented as a 'nop'
  j wfi_loop                // Loops to another wfi in case this HW thread wasn't moved to somewhere else

boot_core:
  sll a0, a0, 2             // a0 = a0 << 2 & b'11111 = a0 << 2                 // offset for hart msip
  add a0, a0, a1            // a0 = a0 + a1
  sw zero, 0(a0)            // *(a0 + 0) = 0                                    // clear the interrupt
  li a0, DRAM_BASE          // a0 = DRAM_BASE                                   // program reset vector
  csrw mepc, a0             // mpec = a0                                        // go from interrupt to start of user program
  csrr a0, mhartid          // a0 = mhartid                                     // hartid for next level bootloader
  la a1, _dtb               // a1 = PC + _dtb
  li a2, 0x80               // a2 = b'10000000                                  // set mstatus MPIE to 0
  csrc mstatus, a2          // mstatus = mstatus & ~a2 = mstatus & b'01111111
  mret                      // instruction for returning from M-Mode trap

_dtb:
  .incbin DEVICE_TREE
```

Listing 8: *SMP-enabled Bootrom from* `fpga-zynq:new-devices`

A.1.2 Generating a multicore rocket chip instance

Rocket Chip is a very configurable processor implementation. By simply changing some configuration parameters, one might easily change, for example, the number of cores or size of caches of the instance to be generated. Specific configuration combinations are aggregated under `Config` classes in `rocket-chip/src/main/scala/chip/Configs.scala`, and new combinations might be easily introduced by the addition of new `Config` classes to the same file.

On top of that, `fpga-zynq` defines a new set of configuration combinations that allow the generation of Rocket Chip instances deployable to Zynq boards. The three main configurations are called `ZynqFPGAConfig`, `ZynqMediumFPGAConfig`, and `ZynqSmallFPGAConfig`. Such combinations extend those defined by `rocket-chip` itself by adding support for communication with the ARM cores from Zynq boards, and they are defined in `fpga-zynq/common/src/main/scala/Configs.scala`.

Given that all these three configurations defined by `fpga-zynq` are single-core and do not generate RoCC extensions, we had to define our own multi-core, RoCC-capable configuration in `fpga-zynq/common/src/main/scala/Configs.scala`. In order to create configurations involving more than two cores, we also had to add new configurations to `rocket-chip/src/main/scala/chip/Configs.scala`, given that by default it does not implement any configurations involving a larger number of cores.

Finally, one might generate a dual-core, RoCC-capable Rocket Chip instance for ZC706 by updating the line containing the `CONFIG` setting in `fpga-zynq/zc706/Makefile` to `CONFIG = ZynqDualCoreFPGAConfigWithRoccExample` and doing:

```
$ make project
$ make fpga-images-zc706/boot.bin
```

Listing 9: *Generating boot artifacts for a dual-core, RoCC-capable Rocket Chip instance for ZC706.*

A.1.3 Generating a SMP-enabled RISC-V Linux image

Once we have a Rocket Chip instance capable of running SMP Linux, we are left with the task of generating the Linux image itself. Doing so is made much easier by the `freedom-u-sdk` project, which provides *Buildroot* configuration files capable of generating SMP-capable Linux images for RISC-V. Moreover, the project includes tools tools for automatically embedding such Linux images in `bbl` artifacts, so that Linux can be easily deployed to RISC-V implementations like Rocket Chip.

Buildroot, which is included in `freedom-u-sdk` as a Git-submodule, is a tool for automating the generation of working Linux images for embedded systems, being similar to Yocto. It allows one to reliably generate a cross-compilation environment targeting the relevant architecture, set adequate Linux build parameters, generate an `initramfs`, and build and include several software packages into that `initramfs`. All settings relevant to a complete Buildroot execution are defined in a small set of configuration files that is particular to a given embedded system target. For convenience, Buildroot comes with several pre-defined sets of configuration files, so that one might use it out-of-the-box for the commonest embedded systems.

Even though `freedom-u-sdk` includes everything necessary for generating Linux images for RISC-V, their versions and configuration files are not necessarily compatible with the Rocket Chip

instances generated by `fpga-zynq`. In particular, we found that the version of `riscv-pk` included in the master branch of `freedom-u-sdk` — as of the time of our development — was not compatible with the version of `riscv-fesvr` that we proved to work with the pre-built `bbl` artifacts included in `fpga-zynq`. Consequently, we decided to (1) make our own fork of `freedom-u-sdk`, in which we could instantiate dependent tools at their most adequate versions; and (2) include `freedom-u-sdk`, together with some other useful work material, in a public Docker image, so that one could more easily deploy and use our Linux-generation environment.

Downloading the Docker image might be accomplished with:

```
$ sudo docker run -it lucashmorais/fpga-env:1.1
```

Listing 10: *Downloading Docker image for generating SMP Linux for RISC-V*

Using that Docker image, one might build a `bbl` artifact including that Linux environment by doing:

```
DOCKER$ make -f Makefile.nondestructive bbl
```

Listing 11: *Generating `bbl` with the Docker image*

Additionally, one might even test that `bbl` artifact on top of a quad-core instance of Spike — the reference RISC-V simulator — by doing:

```
DOCKER$ make -f Makefile.nondestructive sim
```

Listing 12: *Running `bbl` on quad-core Spike*

A.2 Porting the OmpSs runtime library and compiler to the RISC-V architecture.

Porting OmpSs to RISC-V involves (1) cross-compiling the Nanos++ runtime library to RISC-V and (2) building an x86 instance of the Mercurium source-to-source compiler capable of generating RISC-V executables. Both tasks are very straightforward, as doing

```
$ ./configure --prefix=$INSTALLATION_PATH --host=riscv64-unknown-linux-gnu
```

is enough for configuring Nanos++ to be cross-compiled to RISC-V and

```
$ ./configure --prefix=$INSTALLATION_PATH --enable-ompss
↪  --with-nanox=$RISCV_NANOS_INSTALLATION_PATH
↪  --target=riscv64-unknown-linux-gnu
```

makes Mercurium be built with the RISC-V cross-compiler as its backend.

Testing the Nanos++/Mercurium instances generated in this way may be achieved by (1) using the newly-generated Mercurium instance to compile OmpSs benchmarks for RISC-V, (2) packing Nanos++ library files and these test programs into a working Linux `initramfs`, (3) generating

a `bbl` artifact containing that `initramfs`, (4) booting that `bbl` artifact on top of a multi-core instance of Spike, and (5) running the programs in the simulated Linux environment. The `bbl` artifact from step (3) may also be booted on an FPGA-based Rocket Chip softcore.

A.3 Publishing Docker image with tools for building SMP Linux for Rocket Chip.

In order to ease replication of our work, we captured a working snapshot of our software toolchain for generating SMP Linux images for Rocket Chip in a Docker image available at our Docker Hub page [2].

This docker image includes an instance of the `freedom-u-sdk` repository checked-out at our `picos-port` branch, which defines the adequate versions of the `linux`, `riscv-pk`, `riscv-fesvr`, and `riscv-gnu-toolchain` submodules.

This Docker image should let novices generate adequate SMP Linux images and building artifacts for Rocket Chip softcores based on either Ultrascale+ MPSoC or Zynq-7000 much sooner than if they had to replicate the work described so far.

A.4 Creating and testing example RoCC accelerators.

As described before, the RoCC interface of Rocket Chip greatly eases the development of new instructions serviced by custom accelerators. Accordingly, as soon as we finished the setup process just reported, we started exploring how to deploy RoCC accelerators and define their custom instructions. In the following lines, we explain how one might achieve both goals.

Supposing the use of the `fpga-zynq` toolchain for the ZC706 board, generating an instance of Rocket Chip with a new custom accelerator involves three basic actions. First, one should implement the new accelerator in the `LazyRoCC.scala` [3] file; then, one should make sure that the Rocket Chip cores are actually synthesized with the accelerator, which can be achieved, for example, by modifying the `WithRoccExample` class from the `coreplex/Configs.scala` [4] file; finally, one should make sure that the top-level configuration class referred to by `zc706/Makefile` config- extends the `WithRoccExample` just modified.

Once this is done, issuing `$ make project` is enough for re-generating the Verilog files of Rocket Chip with support for the accelerator and updating the Vivado project with it.

After that, one might generate C/C++ programs exercising the new custom instructions with the help of the macros provided by `rocc-software`[5]. If these programs are single-threaded, they might them be easily tested on the FPGA-based Rocket Chip instance with the help of `riscv-pk`.

Before using `riscv-pk` for that, though, it is necessary to make sure that, prior to actually running the binary payload it is given, `riscv-pk` sets the `XS` field of the `mstatus` status register of the relevant core to `0x1`, which can be achieved by changing the definition of the `mstatus_init()` function [6] to the contents of Listing 13.

[2] https://hub.docker.com/r/lucashmorais/fpga-env/
[3] Full path from `rocket-chip` top directory: rocket-chip/src/main/scala/tile/LazyRoCC.scala
[4] Full path from `rocket-chip` top directory: rocket-chip/src/main/scala/coreplex/Configs.scala
[5] https://github.com/IBM/rocc-software
[6] Which can be found at `riscv-pk/machine/minit.c`

```
static void mstatus_init()
{
  // Enable FPU
  write_csr(mstatus, MSTATUS_FS);

  // Enable user/supervisor use of perf counters
  write_csr(scounteren, -1);
  write_csr(mcounteren, -1);

  // Enable XS
  if (supports_extension('X'))
    set_csr(mstatus, (MSTATUS_XS & (MSTATUS_XS >> 1)));

  // Enable software interrupts
  write_csr(mie, MIP_MSIP);

  // Disable paging
  write_csr(sptbr, 0);
}
```

Listing 13: *Modified version of* `mstatus_init` *that automatically sets* `XS` *to* `0x1`.

If one doesn't do so, programs using the custom instructions will lead Rocket Chip to generate illegal instruction traps and abort execution [7].

A.5 Implementing Picos Manager

As described in Section 4.6, *Picos Manager* arbitrates communication between Picos and core-specific RoCC accelerators. Consequently, implementing it involved the following challenges:

- Developing Picos Manager core HDL

  Before being integrated to the rest of the system, Picos Manager had its functionality developed in isolation, with help of Verilator-based testbenches. After every design iteration, we generated the corresponding Verilog source file, processed it with Verilator, and evaluated the current design version using C++ Verilator testbenches and GTKWave. Once tests indicated a fully functional system, we proceeded to integrate it to the rest of the system.

- Instantiating it as a singleton IP in Rocket Chip

  Most IPs in multi-core instances of Rocket Chip are instantiated once for every core. Picos Manager, on the other hand, should be instantiated only once in the whole system. Because of that, we had to find a location in Rocket Chip where a Picos Manager code instantiation would not lead to multiple actual FPGA instantiations.

- Letting the Rocket Chip top-level module expose part of Picos Manager signals

  Picos Manager should be connected to Picos. Given that it is instantiated inside Rocket Chip, its signals should be exposed by the top-level module of Rocket Chip, so that Picos, which is external to the latter, may be connected to Picos Manager.

[7] Unless the alternative strategy described in section is used.

- Exposing core-specific Picos Manager signals to each of the RoCC accelerator instances

  Although RoCC accelerators benefit from RoCC interface for easily connecting to Rocket Chip cores, Rocket Chip does not have any provision for easily connecting RoCC accelerators to custom singleton IPs. Consequently, we had to modify the interfaces between several layers of Rocket Chip sub-modules to connect the single Picos Manager instance, which is located close to the top-level module, to each of the core-specific RoCC accelerators.

A.6 Integrating Picos to Rocket Chip and running binaries on top of the end system

Once Picos Manager and the RoCC accelerator were implemented, Picos integration was straightforward, as it only required connecting Picos to Picos Manager through the top-level signals of the Rocket Chip IP and ensuring proper clocking and reset signaling.

Since Picos features some debug signals for counting the number of tasks that have been submitted, made ready, or executed, we leveraged the ZC706 User LEDs and general purpose buttons for displaying these counters in binary encoding. This was useful for making sure that Picos was actually being used.

For assisting the development of the benchmarks, we extended the Spike RISC-V simulator so as to enable it to simulate our six custom instructions. In order to do so, we just had to modify the `dummy_rocc` example extension provided by `riscv-isa-sim`, creating entries for every instruction and letting them invoke the behavior modeled in `PicosModel.hpp`. The latter source file implements a simplified version of Picos functionality that serializes Task Parallel programs but that is sufficient for the validation of benchmarks targeting the real system.

With such modified version of Spike, one may then execute programs using the custom instructions by invoking:

```
$ ./spike --extension=delegate pk some_test_program
```

In such case, one is using `pk` to handle Linux system calls. As mentioned in Section A.4, in general, one needs to ensure that `pk` sets up the `XS` field of the `mstatus` status register to `0x1`, which can be achieved with the procedure described in that Section.

Alternatively, one could actually boot a full-fledged Linux environment with support to applications using the new instructions with the following:

```
$ ./spike --extension=delegate bbl
```

In both scenarios, *delegate* denotes the name of the Spike extension supporting the custom instructions.

Just as in the previous case, here one must ensure that illegal instruction traps are not going to prevent program execution. Nonetheless, since this case does not involve `pk`, the previous solution does not apply. Instead, one should either (1) make Linux adequately configure `XS` bits or (2) alter Rocket Chip in such a way that RoCC instructions are correctly executed regardless of `XS` settings.

A.6.1 Enabling the execution of RoCC benchmarks from within Linux

While developing our system, we first pursued the first path, which lead us to test several modifications to assembly code called by Linux in early stages with the goal of setting `XS` bits to `0x1`. Nonetheless, since Linux seemed to be overwriting `mstatus` at some later point in time, which was made manifest by the fact that the execution of RoCC benchmarks continued to generate illegal instruction traps regardless of these modifications to boot-related assembly code. That being true, we decided to pursue a solution of the second kind, which involved disabling the generation of illegal instruction traps triggered by incorrect setting of `mstatus`.

By inspecting Rocket Chip Chisel code, we found that the `tile/CSR.scala` source file controlled behavior related to status registers. It also contained code that defined when RoCC instructions could trigger illegal instruction traps, which, according to the default settings, happens every time a RoCC instruction is executed and either the `XS` field of `mstatus` or the `X` field of the *Machine ISA register* (`misa`) are equal to `0x0`. The latter write-any, read-legal register is responsible for describing which ISA extensions should be supported by the RISC-V implementation. Setting the `X` bit of that register (`misa[23]`) allows the execution of instructions belonging to ISA extension *X*, which comprises RoCC instructions.

Suppressing these triggers of illegal instruction traps was then accomplished by commenting the Chisel line that follows, which defines the two conditions just described:

```
io.decode.rocc_illegal := io.status.xs === 0 || !reg_misa('x'-'a')
```

After doing so, regenerating the Verilog files for Rocket Chip, updating the whole-system Vivado project and regenerating the bitstream gave us a new testing environment where the RoCC benchmarks could be successfully run from within Linux.

A.6.2 Embedding benchmarks in Linux initramfs

Creating a `bbl` image containing RoCC benchmarks can be done with the help of the provided Docker image. The `freedom-u-sdk` project included in such image uses *Buildroot* for generating Linux images compatible with Rocket Chip. Consequently, one might add arbitrary files to the `initramfs` of the generated Linux instance with the aid of the `BR2_ROOTFS_OVERLAY` configuration parameter of Buildroot, which describes overlay paths that should have their content directly copied to the generated Linux filesystem.

These overlay directories are merged with the Linux filesystem from the Linux root folder. Thus, one wanting to add a certain `foo.txt` file to `/home/some_user/documents/` should store that file in `$BR2_ROOTFS_OVERLAY/home/some_user/documents/`.

In particular, our provided Docker environment defines `BR2_ROOTFS_OVERLAY="/workdir/nanos-cross-pbin"` in both `freedom-u-sdk/conf/buildroot_initramfs_config` and `freedom-u-sdk/conf/buildroot_rootfs_config` configuration files, so our benchmarks are stored in `/workdir/nanos-cross-pbin/root`.

Generating the Linux image with embedded benchmarks can then be achieved by doing:

```
$ make -f Makefile.nondestructive bbl
```

Re-issuing that command might fail to update the Linux `initramfs`, since modifications to overlay directories do not trigger its regeneration. Consequently, one aiming to sync the Linux

image with the contents of these directories should clean `initramfs` files before building `bbl`. This scenario is exemplified by the following code:

```
$ make clean-filesystems
$ make clean-bbl
$ make -f Makefile.nondestructive bbl
```

A.7 Porting System to Ultrascale+ MPSoC

Most of our initial development was carried out in the context of a research internship at the Barcelona Supercomputing Center, where we had access to a Xilinx ZC706 board. When moving back to our original research institute, though, a board of the same model was not available, so we had to re-target our project to a Xilinx ZCU102-ES2 device. Fortunately, the latter board bears several similarities with the ZC706, which limited the amount of work needed to port our software/-softcore elements to the new environment. Even so, their differences did require modifications to Rocket Chip code and to `riscv-fesvr`. In this section, we aim to explain the nature of these changes.

A.7.1 Main porting issues

The differences between ZC706 and ZCU102-ES2 boards that are most relevant to our work are:

1. the two boards have different memory offsets for the communication ports bridging the ARM processor to the Programming Logic, and
2. their ARM Application processors implement different ISA's.

Regardless of the board used, communication between the ARM host processor and the RISC-V softcore is mediated by `riscv-fesvr`. While this tool is in principle agnostic to the exact communication venue used to bridge the gap between the Application Processor and RISC-V, the `fpga-zynq` project opted to provide low-level communication between these two entities through one of the AXI4 buses between the Application Processor and the Programming Logic. In this toolchain, this capability is provided by a Rocket Chip interface of memory-mapped registers and FIFOs and a user-level driver that knows how to interact with these structures to transfer data — including user I/0 — between these two domains. This Rocket Chip interface is defined by the `ZynqAdapter` cake-pattern Chisel entities defined in `fpga-zynq/common/src/main/scala/ZynqAdapter.scala`, while some of its synthesis parameters are defined by `fpga-zynq/common/src/main/scala/Configs.scala`. The ARM user-level driver that allows `riscv-fesvr` to interact with this Rocket Chip interface is implemented by the sources found in `fpga-zynq/common/csrc` and is called `zynq_driver`.

Since the addresses of these structures are defined with respect to the base physical address of the AXI4 bus used, `zynq_driver` needs to know this base address to access them. In fact, that address is hardcoded in its `zynq_driver.cc` source file. On the other hand, given that, according to Difference (1), the ZCU102-ES2 does not have the same memory offsets as the ZC706, porting our RISC-V FPGA/software tools to the new board requires changing this hardcoded offset from `0x43C00000` to `0xA4000000`. A similar modification should be made in `fpga-zynq/`

`common/src/main/scala/Configs`, since one of the settings it defines for the ZynqAdapter (the `ZynqAdapterBase` option) refers to the same offset.

Moreover, Difference (2) requires that `riscv-fesvr` and `zynq_driver` be cross-compiled to the AARCH64 ISA, which differs from the ARMv7-A ISA implemented by the Application Processor in the ZC706. Compiling `riscv-fesvr` for AARCH64 can be achieved by doing the following from its root path:

```
$ mkdir build && cd build
$ CC=aarch64-linux-gnu-gcc CXX=aarch64-linux-gnu-g++ ../configure
↪  --host=aarch64-unknown-linux-gnu
$ make
```

Cross-compiling `zynq_driver` for AARCH64 can be effected in a similar way.

A.7.2 Additional challenges

Apart from that issues just discussed, porting the system to the new environment also requires one to learn how to generate ZCU102-ES2 boot artifacts containing working Linux images and the Rocket Chip bitstream. It is also highly desirable that the ARM Linux environment should have support for SSH, networking based on static local IP, and programs using `libstdc++`. Furthermore, optimizing terminal input speed for `riscv-fesvr` is also valuable, as our previous experience with the ZC706 showed that the RISC-V Linux terminal input could be exceedingly slow for the default parameters of `ZynqAdapter`.

Generating Linux images for the ZCU102-ES2 board might be accomplished by using the Xilinx `petalinux` toolchain. Creating Linux instances without the aid of `petalinux` is possible, but can be tedious and time-consuming, as `petalinux` automates the generation of an adequate device-tree tailored for any given board with any given boot parameters; provides means for easily deploying the Linux image, the bitstream and other essential boot artifacts through JTAG; allows the generation of boot SD cards, etc.

While `petalinux` supports many different workflows and configurations, the procedure that we followed for building Linux for the board with the requirements above was the following:

- Increasing depth of `ZynqAdapter` FIFOs

  Data exchanged between the ARM substrate and the RISC-V softcore go through the FIFOs defined by `ZynqAdapter`. Consequently, it is reasonable to suppose that the performance of their communication is influenced by the depth of these FIFOs, as `riscv-fesvr` limits the frequency at which these FIFOs might be queried and/or updated. Our experience shows that increasing the depth of these structures from 16 (the default value) to 512 delivers noticeable I/O improvements — leading to faster input to the RISC-V terminal provided by `riscv-fesvr` — without substantially increasing the area occupied in the FPGA by the overall Rocket Chip design.

  Effecting this change might then be accomplished by changing the `SerialFIFODepth` setting of the `WithZynqAdapter` class in the `fpga-zynq/common/src/main/scala/Configs.scala` file, regenerating Verilog files, updating the Vivado project, and regenerating the bitstream.

- Using Vivado, generating a Hardware Description File (HDF) for the ZCU102-ES2 project

  When used for managing Zynq-7000 and Ultrascale+ MPSoC projects, which often involve interplay between a hardware design sitting in the Programming Logic and auxiliary software applications running on the Application Processor, Vivado allows one to generate corresponding Hardware Description Files. A HDF describes, among other things, the address map of the hardware design and information about the peripherals enabled for the PL-AP SoC. Such information is invaluable for the synthesis of a Linux environment compatible with the environment defined by the HDF, as Linux must have information about a great number of devices — offsets and parameters serial and ethernet ports, size and offset of main memory, etc — to be successfully booted.

  These HDF files can be used for creating custom `petalinux` projects tailored for the exact settings of a given PL+AP project.

- Downloading the default Board Support Package (BSP) for ZCU102-ES2 from Xilinx repositories

  While one of the strongest features of `petalinux` is its ability to automatically create Linux build and boot configuration artifacts tailored to specific boards with specific configurations, it might also be used for generating and deploying sample working Linux environments for Xilinx AP+PL boards. These sample environments give the user the opportunity to exercise many functionalities of a specific board without having to go through the process of creating a Vivado project, generating a HDF file, creating a `petalinux` project based on that HDF; and building Linux from scratch. This quick-test functionality is provided by sample Board Support Packages, which can be downloaded from Xilinx. A `petalinux` project created from a BSP can be later re-targeted to a specific board configuration by the automated processing of the relevant HDF file.

  Even though our development effort required the creation of a `petalinux` tailored to our Vivado project containing Rocket Chip, making such a project from a BSP instead of generating it from scratch lets one use the pre-built images contained in the BSP to perform several useful health checks on the board. Consequently, we opted to generate our `petalinux` project from the `xilinx-zcu102-zu9-es2-rev1.0-v2017.3-final.bsp` made available by Xilinx.

  Creating a working Linux environment from that BSP can be accomplished with the following:

```
$ petalinux-create -t project -s
↪  xilinx-zcu102-zu9-es2-rev1.0-v2017.3-final.bsp
```

  After that, a project named `xilinx-zcu102-zu9-es2-rev1.0-v2017.3-final` will have been generated. Then, one should be able to deploy the pre-built image to the board through JTAG by doing:

```
$ petalinux-boot --jtag --prebuilt 3
```

  It is also possible to locally emulate the same image using QEMU doing the following:

```
$ petalinux-boot --qemu --prebuilt 3
```

Then, configuring a custom Linux image, building it and deploying it to the board through JTAG can be effected with:

```
$ petalinux-config
$ petalinux-build
$ petalinux-boot --jtag --kernel
```

It should be noticed that the custom environment thus created still does not include any custom bitstream nor takes into account a custom system address map. As we will show next, compatibility with a custom address map requires the reconfiguration of the project with respect to a relevant HDF, while programming the board with a custom bitstream requires either importing that bitstream to the project or explicitly referencing it while calling the `petalinux-boot` tool.

- Updating the `petalinux` project with the new HDF

  After creating a boilerplate project from the ZCU102-ES2 BSP as just indicated, one might re-target it to the Vivado project containing Rocket Chip as follows:

```
# Supposing that the HDF is available at the current path
petalinux-config --get-hw-description=rocketchip_wrapper.hdf
```

  At this point, one should already be able to build a working Linux environment using the Rocket Chip bitstream. Nonetheless, we chose to only doing so after performing additional configurations.

- Using `petalinux-config` to enable static IP addressing

  Enabling static IP addressing for the board eases the process of connecting to it through SSH, since it allows one to directly connect a computer to it through ethernet without the need to configure a DHCP server specific to the link.

  Doing so might be easily accomplished with the help of `petalinux-config`, which launches a `menuconfig` environment that allows the user to set several Linux build parameters. To this effect, after calling the tool at the root path of the `petalinux` project, one should simply navigate to `SubsystemAUTOHardwareSettings/EthernetSettings` and uncheck `Obtain IP address automatically`. The same page also lets one configure several relevant options for static IP addressing.

- Using `petalinux-config -c rootfs` to enable `libstdc++`

  The `riscv-fesvr` tool is dynamically linked to `libstdc++`. Consequently, one must make sure that the ARM Linux environment where it is going to be run contains that dynamic library.

  Making this package be built and installed in the filesystem of the Linux image can be effected by calling `petalinux-config` from the root path of the `petalinux` project, navigating to `FilesystemPackages/misc/gcc-runtime/libstdc++`; and checking `libstdc++`.

- Building and testing the system

Finally, supposing that the bitstream generated by the Vivado has been copied to the root path of the `petalinux` project, building and testing the system might be accomplished by doing:

```
$ petalinux-config
$ petalinux-build
$ # Supposing that the bitstream is named 'rocket_chip_bitstream.bit'
$ petalinux-boot --jtag --kernel --fpga --bitstream
↪  rocket_chip_bitstream.bit
```

Bibliography

Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H., and Waterman, A. (2016). The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley. 2, 3, 4

Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K. (2012). Chisel: constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221. IEEE. 2, 3

Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E., and Labarta, J. (2004). Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP*, volume 8, page 56. 1, 37

Bienia, C. and Li, K. (2009). Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, volume 2011. 37

Celio, C., Chiu, P.-F., Nikolic, B., Patterson, D. A., and Asanović, K. (2017). Boom v2: an open-source out-of-order risc-v core. Technical Report UCB/EECS-2017-157, EECS Department, University of California, Berkeley. 3

Celio, C., Patterson, D. A., and Asanović, K. (2015). The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical Report UCB/EECS-2015-167, EECS Department, University of California, Berkeley. 40

Dallou, T., Elhossini, A., and Juurlink, B. (2013). FPGA-based prototype of nexus++ task manager. In *6th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS) 2013, Co-located with SC 2013*. 6

Dallou, T., Engelhardt, N., Elhossini, A., and Juurlink, B. (2015). Nexus#: A distributed hardware task manager for task-based programming models. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1129–1138. IEEE. 6, 7

Dallou, T. and Juurlink, B. (2012). Hardware-based task dependency resolution for the starss programming model. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 367–374. IEEE. 1

Dallou, T., Lucas, D. C. S., Araujo, G., Morais, L., Barbosa, E. F., Frank, M., Bagley, R., and Sayana, R. (2016). Task parallel programming model+ hardware acceleration= performance advantage. In *Hot Chips 28 Symposium (HCS), 2016 IEEE*, pages 1–1. IEEE. 6

Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011). OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193. 1

Duran, A., Perez, J. M., Ayguadé, E., Badia, R. M., and Labarta, J. (2008). Extending the OpenMP tasking model to allow dependent tasks. In *International Workshop on OpenMP*, pages 111–122. Springer. 6

Etsion, Y., Cabarcas, F., Rico, A., Ramirez, A., Badia, R. M., Ayguade, E., Labarta, J., and Valero, M. (2010). Task superscalar: An out-of-order task pipeline. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 89–100. IEEE. 1, 6

GNU Foundation (2005). An OpenMP Implementation for GCC. http://gcc.gnu.org/projects/gomp. 1

Intel Corporation (2013). Intel OpenMP Runtime Library. https://www.openmprtl.org. 1

Kumar, S., Hughes, C. J., and Nguyen, A. (2007). Carbon: architectural support for fine-grained parallelism on chip multiprocessors. *ACM SIGARCH Computer Architecture News*, 35(2):162–173. 6

Meenderinck, C. and Juurlink, B. (2010). A case for hardware task management support for the starss programming model. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 347–354. IEEE. 6

Sinnen, O., Pe, J., and Kozlov, A. V. (2007). Support for fine grained dependent tasks in OpenMP. In *International Workshop on OpenMP*, pages 13–24. Springer. 6

Tan, X. (2018). Hardware runtime management for task-based programming models. 1, 11

Tan, X., Bosch, J., Jiménez-González, D., Álvarez-Martínez, C., Ayguadé, E., and Valero, M. (2016). Performance analysis of a hardware accelerator of dependence management for task-based dataflow programming models. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 225–234. IEEE. 6, 7

Tan, X., Bosch, J., Vidal, M., Álvarez, C., Jiménez-González, D., Ayguadé, E., and Valero, M. (2017). General purpose task-dependence management hardware for task-based dataflow programming models. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 244–253. IEEE. 1, 6, 7, 11, 40

Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., and Gautier, T. (2014). Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In *International Workshop on OpenMP*, pages 16–29. Springer. 37

Wang, C., Li, X., Zhang, J., Chen, P., Chen, Y., Zhou, X., and Cheung, R. C. (2015). Architecture support for task out-of-order execution in MPSoCs. *IEEE Transactions on Computers*, 64(5):1296–1310. 6, 7

Wang, C., Li, X., Zhang, J., Zhou, X., and Nie, X. (2013). MP-Tomasulo: A dependency-aware automatic parallel execution engine for sequential programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(2):9. 6

Waterman, A., Lee, Y., Patterson, D. A., Asanović, K., level Isa, V. I. U., Waterman, A., Lee, Y., and Patterson, D. (2014). The RISC-V Instruction Set Manual. 3

Yazdanpanah, F., Álvarez, C., Jiménez-González, D., Badia, R. M., and Valero, M. (2015). Picos: A hardware runtime architecture support for OmpSs. *Future Generation Computer Systems*, 53:130–139. 1, 6, 7, 11