Adding Tightly-Integrated Task Scheduling Acceleration to a RISC-V Multi-core Processor

Lucas Morais,
Vitor Silva, Alfredo Goldman
University of São Paulo
{moraislh,vitors,gold}@ime.usp.br

Carlos Alvarez, Jaume Bosch
Barcelona Supercomputing Center
{carlos.alvarez,jbosch}@bsc.es

Michael Frank
MagiCore Inc.
michael@pil.net

Guido Araujo
University of Campinas
{guido}@ic.unicamp.br

***Abstract*—Task Parallelism is a parallel programming model that provides code annotation constructs to outline tasks and describe how their pointer parameters are accessed so that they might be executed in parallel, and asynchronously, by a runtime capable of inferring and honoring their data dependence relationships. It is supported by several parallelization frameworks, as OpenMP and StarSs.**

Overhead related to automatic dependence inference and to the scheduling of ready-to-run tasks is a major performance limiting factor of Task Parallel systems. To amortize this overhead, programmers usually trade the higher parallelism that could be leveraged from finer-grained work partitions for the higher runtime-efficiency of coarser-grained work partitions. Such problems are even more severe for systems with many cores, as the task spawning frequency required for preserving cores from starvation grows linearly with their number.

To mitigate these problems, researchers have designed hardware accelerators to improve runtime performance. Nevertheless, the high CPU-accelerator communication overheads of these solutions hampered their gains.

We thus propose a RISC-V based architecture that minimizes communication overhead between the HW Task Scheduler and the CPU by allowing Task Scheduling software to directly interact with the former through custom instructions. Empirical evaluation of the architecture is made possible by an FPGA prototype featuring an eight-core Linux-capable Rocket Chip implementing such instructions.

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.

***Index Terms*—Task Scheduling, Rocket Chip, RoCC Interface, Picos, RISC-V, Chisel.**

MICRO '52, October 12–16, 2019, Columbus, OH, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6938-1/19/10...$15.00
https://doi.org/10.1145/3352460.3358271

I. Introduction

For many of todays parallel programming tools, ease of use often comes 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 [12] that Task Parallelism can be understood as a technique for semi-automatically transforming sequential imperative programs into dataflow representations, for which elementary operations are performed concurrently and asynchronously, as soon as their input data become available. In fact, as others pointed out [12], task parallelism does for tasks the same that Tomasulo's Algorithm does for instructions — it infers data relationships between them and lets them be run by 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 [3, 10, 13, 14]. 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, insufficient scheduling throughput of the software runtime leads to part of the cores becoming starved 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 acceleration, leading to largely successful designs [8, 18, 20, 24]. For example, the Picos [20] Task Scheduling accelerator 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.

This paper proposes an architecture that tightly integrates a Task Scheduling HW accelerator with a general purpose CPU to minimize communication latency, reduce runtime overhead, and consequently increase the performance of applications parallelized with Task Scheduling.

A. Contributions

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 and made visible to applications through custom instructions. 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 [2] to integrate Picos, a mature Task Scheduling accelerator, to Rocket Chip, an open-source, silicon-proven, multi-core implementation of RISC-V [1].

To measure Task Parallel application performance of this platform, we adapted Nanos — a software-only Task Scheduling runtime, targeting the OmpSs programming model — to our system. We also developed a new lightweight, high-performance Task Scheduling runtime called *Phentos* from scratch, building upon the lessons learned while adapting Nanos.

The semantics of the custom instructions are also general enough that different HW Schedulers could be integrated.

As discussed in Section VI, the increased maximum task scheduling throughput of our system gives it two closely related advantages over previous solutions. If (1) mean task granularity is kept constant, Task Scheduling programs might be efficiently scheduled to a larger number of cores, and (2) if the number of cores is kept constant, workloads using smaller tasks than ever before might 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%) occurring for only 1 of the 37 analyzed workloads, involving 5 different programs.

B. Paper structure

This paper is divided as follows: Section II presents our approach in the context of related work; Section III describes background material for the following sections; Section IV details the proposed hardware architecture; Section V describes the software systems developed for letting the new architecture serve Task Scheduling applications; Section VI discusses the experimental setup and its results; at last, we present our final remarks in Section VII.

II. 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 of processor extensions for improving scheduling of dependence-less tasks. Then, as StarSs and later OpenMP 4.0 introduced tasks with data dependencies [11, 17], new architectures were proposed for reducing task graph management overhead, and HDL implementations of several of these architectures were conceived [7, 9, 22, 23, 24].

Kumar et al. [15] developed hardware task queues that could be used for accelerating the dynamic scheduling of tasks with only parent/child dependencies. Their work also aimed at improving load balance between cores by implementing a fast HW work stealing mechanism.

Meenderinck and Juurlink [16] 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. [12] proposed the Task Superscalar architecture, which aimed to improved 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. [7] 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++ [6]. 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. [24] 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. [19] 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. [19] 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. [20] 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 [7, 22, 24] 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 [19] or Picos++ [20]. 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 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 enables program execution on 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 have been 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 [20]. In doing so, it greatly expands the number of applications that can be efficiently handled by Task Scheduling.

III. Background

A. 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)

B. Rocket Chip

Rocket Chip is a parametrizable, open-source, Chisel-based SoC generator of RISC-V systems capable of emitting synthesizable RTL code [1]. 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.

C. 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.

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 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.

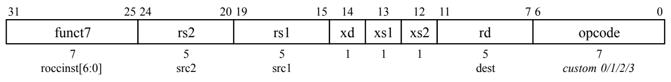


Figure 1. Format of RoCC instructions.

D. Terminology

This subsection 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, 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.

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.

IV. Picos + RISC-V Integration

A. 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. Additionally, Picos Manager decouples the CPU from the API of the HW scheduler. As a consequence, the same architecture with the same custom instructions could be used for integrating

HW schedulers other than Picos. A simplified view of this architecture is given by Figure 2.

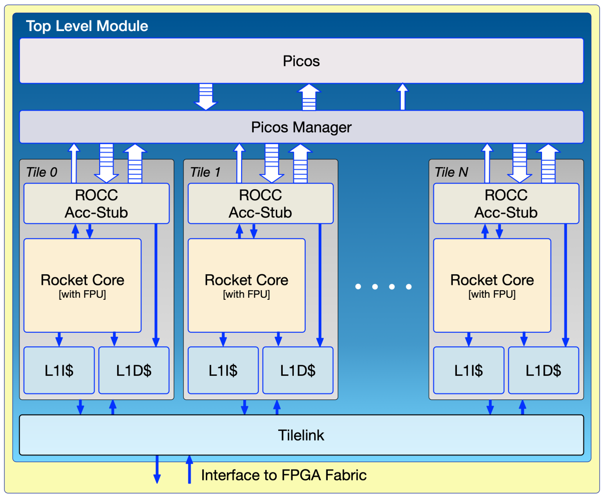


Figure 2. Overview of the Picos + Rocket Chip system architecture

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 2). These instances 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* for developing optimizations like task-scheduling-aware cache prefetching, faster parameter passing for tasks, etc.

B. 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

Name	Description
Submission Request	Informs the system that the core executing this instruction will attempt to submit a task.
Submit Packet	Submits a single 32-bit wide submission packet.
Submit Three Packets	Submits three 32-bit wide submission packets.
Ready Task Request	Requests the system to move one more Ready Task packet from the global Ready Queue to the queue of the executing core.
Fetch SW ID	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	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	Informs the system about the retirement of the task with the Picos ID given.

Table I
SUPPORTED CUSTOM TASK SCHEDULING INSTRUCTIONS.

after the corresponding transaction between Picos 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 might not be 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 IV-C. On the other hand, the *Retire Task* instruction was designed as blocking because (1) Picos is always ready to receive new retirement signals, making the capability of reporting failures useless; and because (2) this reduces compiler register pressure, as the non-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 Figure 3.

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, as Figure 3 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.

C. 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 instruction

As before, let us suppose that 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. 5) 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 the *RoCC Ready Queue*, the routing queue will never be depleted — since there are no ready tasks to distribute — and the *Ready Task Request* being executed by T will never return. Ready tasks 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, though.

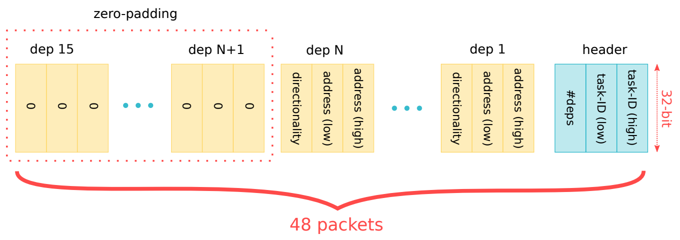


Figure 3. Picos encoding of a task with N data dependencies. This shows that any task is described by $3 * (15 + 1) = 48$ packets. In case a task has N data dependencies — with $0 \leq N \leq 15$ — the last N_z of these packets will be equal to zero, where $N_z = (15 - N) * 3$. In our system, only the first $48 - N_z$ packets must be submitted by the Task Scheduling Runtime to Picos Manager, as zero packets appended by the Submission Handler.

D. Picos

Picos [18, 20, 24] 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.

We chose to use Picos instead of Picos++ because (1) we had prompt access to Picos but not to Picos++; (2) they should display exactly the same performance in our system, given that our CPU-Scheduler communication scheme effectively replaces the asynchronous communication module that made Picos++ faster than Picos; and (3) we were not interested in exploring nested task support, which Picos++ implements and Picos does not. At the same time, replacing Picos with Picos++ should be straightforward, given that they have the same HW interface. Also, for making sure that the gains we report reflect advantages over the previous state-of-the art, the new proposed architecture is compared with the best-performing system based on Picos++ from the literature [20].

E. 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.

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 3). Such null packets are automatically generated by Picos Manager after the relevant Picos Delegate sends the last non-zero packet.

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.

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) 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.

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.

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.

7) Retire Task: The Picos Delegate fulfils *Retire Task* instructions by pushing the payload of the operand register to the Round Robin Arbiter in Picos Manager (see Figure 5). 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 Round Robin Arbiter.

F. Picos Manager

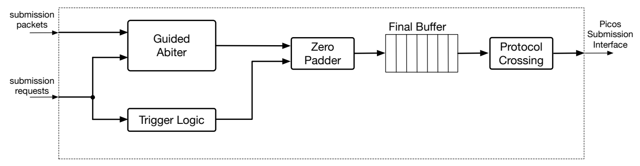


Figure 4. Block diagram of the Submission Handler, a module instantiated by Picos Manager for carrying out transmission of new task descriptors to Picos.

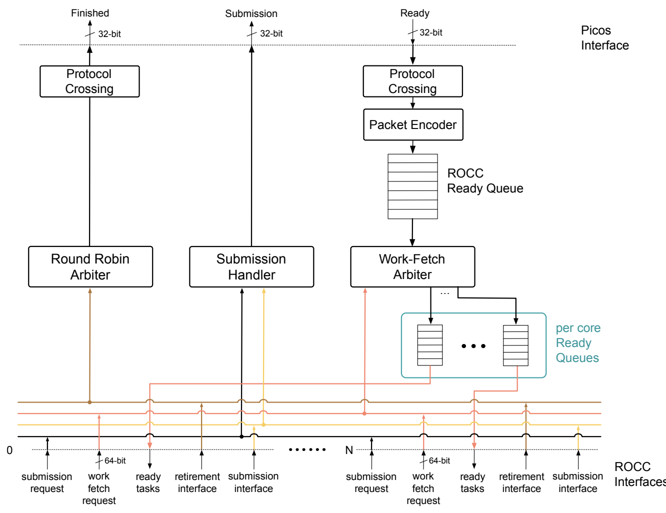


Figure 5. Internals of the Picos Manager module.

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.

1) Interface: As shown by Figure 2, 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 2) includes (9) a 4-bit output signal encoding errors (omitted from figures).

2) Structural elements: As described by Figure 5, 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 by Figure 4 — 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 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 core-specific submission buffers only after the last transmission sequence 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.

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, which is then stored in a central Ready-Task queue.

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 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 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*).

V. 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.

A. 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 dependences 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 functions 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 VI 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 new architecture.

B. The Phentos Fly-Weight Runtime

Phentos was thus 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 dependences, 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 `swID` corresponding to such element. Threads obtain such identifiers by running *Fetch SW ID* instructions as described in Section IV. 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 work-fetch failures have 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, with N varying from 10 to 100 depending on the `taskwait` method being used by the Task Parallel application. 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).

VI. Experimental Evaluation

In this section 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.

Verifying the two first hypotheses implies that the architecture described in Section IV 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.

A. 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-subsection VI-A2. 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-subsection VI-A1.

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 [3], 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.

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 [4] 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 [21].
3) The *stream-deps* and the *stream-barr* programs are micro-benchmarks that evaluate 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.

[1]https://pm.bsc.es/gitlab/benchmarks/parsec-ompss

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.

B. Results and Analysis

1) Comparing Nanos-SW, Nanos-RV, and Phentos: Figure 9 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 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.

2) Deriving theoretical speedup bounds from MTT: As described in Subsection III-D, Maximum Task Throughput (MTT) is the maximum number of tasks from a specific uniform 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:

[2]https://github.com/bsc-pm/ompss-ee

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

where T_{exec} is the fixed task size 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.

Consequently, one might derive a speedup bound MS for that system as a function of mean task size as the following:

$$MS(t) = K \times t$$

Considering that $K = \frac{1}{L_o}$, where L_o is the mean Task Scheduling overhead experienced by tasks during their whole lifetime, MS might then be defined as a function of L_o and T_{exec} as the following:

$$MS(L_o, t) = \frac{t}{L_o} \tag{1}$$

Having this in mind, for four different workloads, we measured the mean Task Scheduling overhead of Nanos-RV and Phentos, as shown by Figure 7. That figure also compares with the previous 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 [20].

Figure 7 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. Such measurements were taken with two different lifetime-overhead-measuring benchmarks: *Task Free*, which generates independent tasks with any number of monitored pointer parameters 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 monitored pointer parameters similarly ranging from 0 to 15.

Based on the figures for the *Task-Chain (1 dep)* case and on Equation 1, we might then evaluate maximum speedup bounds for the various different Task Scheduling platforms as a function of mean task size as shown by Figure 6. That figure shows that the reduced lifetime overheads of Phentos substantially improves 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) Effects of task granularity: As discussed in Subsubsection VI-B2, 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 8 shows how the speedup of Task Scheduling platforms with respect to lower-MTT platforms

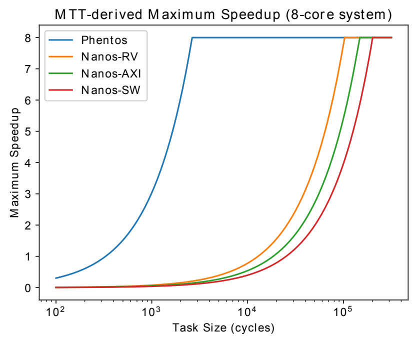
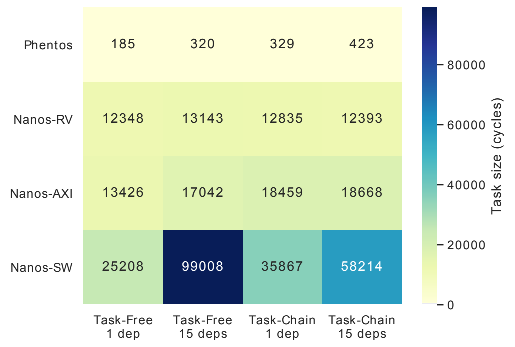


Figure 6. Theoretical MTT-derived speedup bounds for several Task Scheduling platforms with eight cores.



	Task-Free 1 dep	Task-Free 15 deps	Task-Chain 1 dep	Task-Chain 15 deps
Phentos	185	320	329	423
Nanos-RV	12348	13143	12835	12393
Nanos-AXI	13426	17042	18459	18668
Nanos-SW	25208	99008	35867	58214

Figure 7. Lifetime Task Scheduling overhead for several platforms, in Rocket Chip equivalent cycles. The Nanos-RV and Phentos platforms correspond to work described in this paper, while data for Nanos-AXI derives from a recent work based on Picos++ [20]. 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. [5]. Consequently, the Nanos-AXI figures here reported are about 57% higher than those described by Tan et al. [20].

and corresponding serial executions depends on mean task size. The data points there represented correspond to the same benchmark executions reported by Figures 9 and 10.

4) Resource utilization: Table II showcases the resource utilization of several relevant system components. In particular, it shows that the whole Task Scheduling sub-system (including Picos, Picos Manager, and the Delegates) takes less than 2% of the resources of the whole octa-core SoC. Given that the CPU cores are in-order and relatively simple, it is to expect that the same set of HW modules would take an even lower fraction of a production-grade SoC featuring out-of-order cores with a more complete cache hierarchy.

Module	**Usage**	**Fraction**	**Description**
top	384K	100.00%	Whole system
Core	44K	11.56%	Core with FPU and L1$
fpuOpt	18K	4.77%	Floating-point unit
dcache	6K	1.57%	D-cache of a single core
icache	1K	0.32%	I-cache of a single core
SSystem	7K	1.79%	Picos, Picos Manager, and Delegates

Table II
RESOURCE USAGE BREAKDOWN IN NUMBER OF FPGA CELLS.

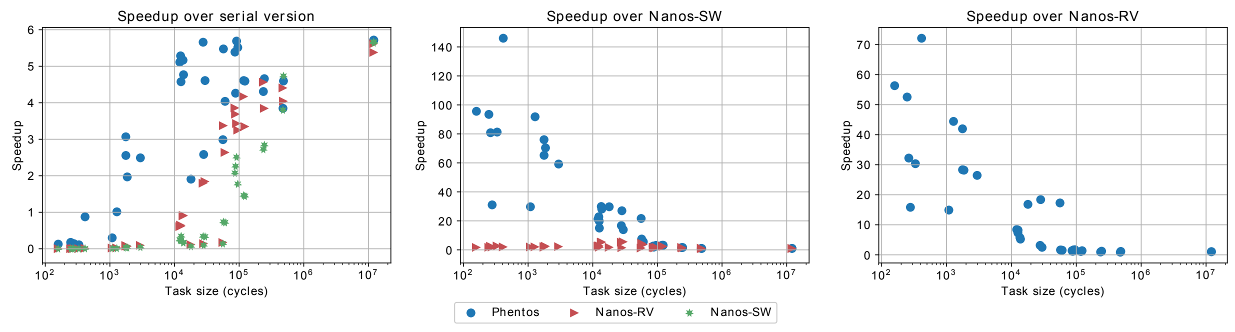


Figure 8. Speedups of programs running on each of the platforms with respect to corresponding serial executions or equivalent runs on lower-MTT platforms.

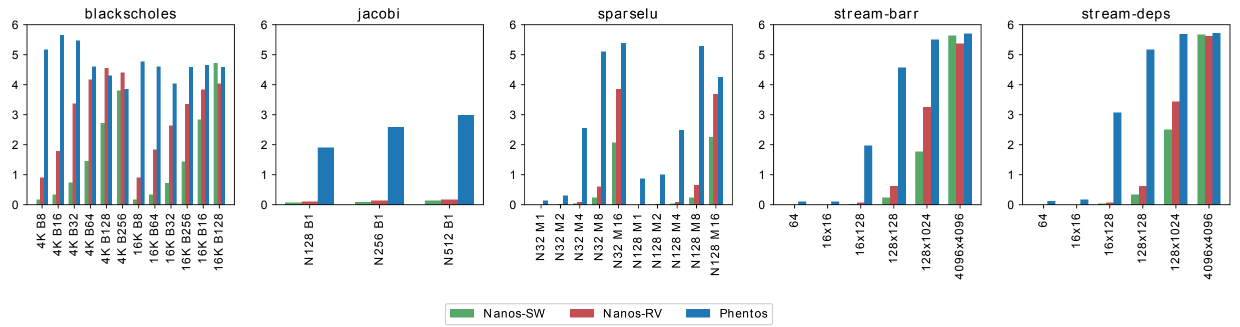


Figure 9. Normalized benchmark performance for all considered inputs and runtimes.

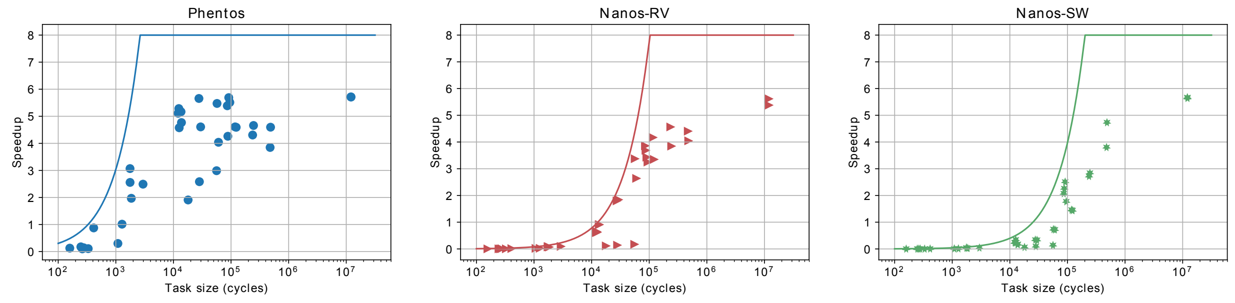


Figure 10. Experimental speedup data of Task Scheduling applications over corresponding serial executions compared with theoretical MTT-derived bounds. As it is the case for Figure 6, MTT values are derived from Task-Chain executions involving one monitored pointer parameter par task.

VII. Conclusion

In this paper, 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 308x by Phentos.

VIII. Acknowledgements

This work was supported by the Spanish Government (projects SEV-2015-0493 and TIN2015-65316-P), the Generalitat de Catalunya (2017-SGR-1414 and 2017-SGR-1328), FAPESP (grants 2017/02682-2, 2018/00687-0, and 2014/25694-8), CNPq (grant 408782/2016-1), and CAPES. We would also like to thank Xubin Tan for her work on Picos.

REFERENCES

[1] 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] 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.

[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.

[4] 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.

[5] 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.

[6] 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*.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] Intel Corporation (2013). Intel OpenMP Runtime Library. https://www.openmprtl.org.

[15] 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.

[16] 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.

[17] 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.

[18] Tan, X., Bosch, J., Alvarez, C., Jiménez-González, D., Ayguadé, E., and Valero, M. (2019). A hardware runtime for task-based programming models. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.