Enabling HW-Based Task Scheduling in Large Multicore Architectures

Lucas Morais, Carlos Álvarez, Daniel Jiménez-González, Juan Miguel de Haro, Guido Araujo, Michael Frank, *Senior Member, IEEE*, Alfredo Goldman, *Senior Member, IEEE*, and Xavier Martorell, *Member, IEEE*

***Abstract*—Dynamic Task Scheduling is an enticing programming model aiming to ease the development of parallel programs with intrinsically irregular or data-dependent parallelism. The performance of such solutions relies on the ability of the Task Scheduling HW/SW stack to efficiently evaluate dependencies at runtime and schedule work to available cores. Traditional SW-only systems implicate scheduling overheads of around 30K processor cycles per task, which severely limit the (*core count, task granularity*) combinations that they might adequately handle. Previous work on HW-accelerated Task Scheduling has shown that such systems might support high performance scheduling on processors with up to eight cores, but questions remained regarding the viability of such solutions to support the greater number of cores now frequently found in high-end SMP systems. The present work presents an FPGA-proven, tightly-integrated, Linux-capable, 30-core RISC-V system with hardware accelerated Task Scheduling. We use this implementation to show that HW Task Scheduling can still offer competitive performance at such high core count, and describe how this organization includes hardware and software optimizations that make it even more scalable than previous solutions. Finally, we outline ways in which this architecture could be augmented to overcome inter-core communication bottlenecks, mitigating the cache-degradation effects usually involved in the parallelization of highly optimized serial code.**

***Index Terms*—Parallel programming, hardware acceleration, Task Scheduling, RISC-V, custom ISA, FPGA.**

Manuscript received 19 January 2023; revised 19 June 2023; accepted 8 August 2023. Date of publication 12 October 2023; date of current version 22 December 2023. This work was supported in part by the TEXTAROSSA project G.A. n.956831, as part of the EuroHPC initiative, in part by the Spanish Government under Grants PCI2021-121964, TEXTAROSSA; PDC2022-133323-I00, Multi-Ka; PID2019-107255GB-C21 MCIN/AEI/10.13039/501100011033; and CEX2021-001148-S, in part by the Generalitat de Catalunya under Grant 2021 SGR 01007, and in part by the FAPESP under Grant 2019/26702-8. Recommended for acceptance by F. Cappello. *(Corresponding author: Lucas Morais.)*

Lucas Morais, Carlos Álvarez, Daniel Jiménez-González, Juan Miguel de Haro, and Xavier Martorell are with the Barcelona Supercomputing Center, 08034 Barcelona, Spain and also with Universitat Politècnica de Catalunya, Carrer de Jordi Girona, 08034 Barcelona, Spain (e-mail: lucas.morais@bsc.es; daniel.jimenez@bsc.es; carlos.alvarez@bsc.es; juan.deharoruiz@bsc.es; xavier.martorell@bsc.es).

Guido Araujo is with the University of Campinas, Campinas-SP 13083-852, Brazil (e-mail: guido@unicamp.br).

Michael Frank is with Arteris Inc., Campbell, CA 95008 USA (e-mail: michael@pil.net).

Alfredo Goldman is with the University of São Paulo, Sao Paulo-CEP 05508-090, Brazil (e-mail: gold@ime.usp.br).

Digital Object Identifier 10.1109/TC.2023.3323781

I. Introduction

PARALLEL programming is widely considered to be more challenging than sequential software development. Both correctness and high performance are difficult to achieve in parallel programs, and validating parallel code can be particularly challenging due to the added indeterminism arising from multiple simultaneous workers. This can lead to low-probability but critical errors that may not be identified during preliminary testing. Such challenges are further compounded by the use of relaxed memory models in many modern high-performance parallel architectures such as ARM, PowerPC, and RISC-V.

Achieving high performance requires evenly partitioning computation among workers across the whole application execution, such that idleness is minimized. Workloads such as matrix multiplication, n-body analysis, and much of image processing can be made to comply with this requirement without much effort, but applications not falling into the data parallel category can have control-flow constraints that hamper their ability of being partitioned into program segments (function calls, loop iterations, etc) taking about the same amount of compute time. Keeping multiple compute units busy with such irregular program segments might then require more complex coordination, such that compute units get dynamically fed with available work as they finish their previous assignment.

Additionally, maximizing the utilization of all available compute units may demand not only correct and efficient work coordination, but also that such work be partitioned as finely as possible, such that, at any moment during the program execution, there are enough pieces of available work to feed all units. Nevertheless, fine-grained work partitioning poses its own threats to application performance, since it amplifies data traffic and related issues. The work in [1], for instance, illustrates both the potential and hurdles of exploring fine-grained parallelism in a HW-accelerated graph mining workload, with substantial speedups being achieved only after great care had been taken to minimize greater orchestration and communication overheads.

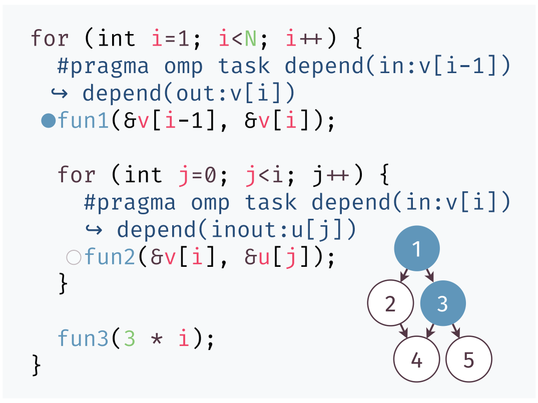
Several parallel programming frameworks were proposed to improve programmer productivity under the previously discussed constraints. Such frameworks might vary substantially in their level of abstraction, target hardware, and supported programming models, as discussed next. Pthreads [2], MPI [3], and CUDA [4] offer fine-grained control over data and computation distribution, but require ad hoc implementation of work distribution logic for each application [5], [6]. Frameworks such as OpenMP [7], Intel oneAPI [8], and OmpSs-2 [9] map more abstract parallel constructs to these lower-level libraries, making applications easier to validate. Other frameworks, such as TensorFlow [10] and PyTorch [11], offer even higher levels of abstraction but are specific to certain application domains, such as machine learning. More abstract frameworks often make it easier for programmers to validate their applications by reliably implementing lower-level mechanics that would otherwise need to be developed and debugged.

The OpenMP, oneAPI, and OmpSs-2 frameworks support various parallel programming models, including OpenMP's concise data-parallel constructs, oneAPI's pipeline and graph-based parallelism, and OmpSs-2's dynamic Task Parallelism. The latter programming model allows data dependency relationships between tasks to be detected at runtime, permitting available workers to execute tasks in parallel in any order that respects these relationships. This allows for more thorough parallelism exploitation than pipeline or traditional graph-based models, as there are parallelization opportunities that only occur for specific inputs and must be handled conservatively by programming models that make decisions at compile time.

While programmer-defined data and computation distribution schemes require minimal runtime overhead, making dynamic decisions during execution can lead to significant penalties. These penalties can include tens of thousands of cycles for every scheduled task [12] and additional memory-related delays due to increased instruction cache pressure or the use of data synchronization barriers in relaxed-memory systems. As a result, dynamic Task Scheduling may not be as effective as other parallelization strategies in certain scenarios. The more workers available, the larger the tasks must be to ensure cores receive a new task before the previous one has finished, avoiding idleness.

Previous work filled that literature gap by providing evidence that HW-based Task Scheduling can dramatically outperform SW-only solutions at scheduling fine-grained tasks to general purpose CPUs, showing that a HW Scheduler tightly integrated to a 8-core processor can reduce scheduling overheads by up to 300x with respect to that baseline, leading to substantial applications speedups. Still, such core count was somewhat limited compared to what can be found in high-end SMP systems. The present work thus builds upon that foundation to deliver comparable advantages for larger processors, and describes further optimizations that make this new version even more capable than its original form. Alongside these extensions, this paper essentially contains the following main contributions:

- the description of a hardware architecture providing low-latency access to HW-accelerated Task Scheduling through custom instructions, bypassing DMA and OS-driver overheads;
- Phentos, a purpose-built, lightweight, user-level software runtime providing highly efficient access to hardware Task Scheduling acceleration;



```
for (int i=1; i<N; i++) {
  #pragma omp task depend(in:v[i-1])
  ↪ depend(out:v[i])
 ●fun1(&v[i-1], &v[i]);

  for (int j=0; j<i; j++) {
    #pragma omp task depend(in:v[i])
    ↪ depend(inout:u[j])
   ○fun2(&v[i], &u[j]);
  }

  fun3(3 * i);
}
```

Fig. 1. Sample code leveraging OpenMP 4.0 task constructs.

- a roofline model for Task Scheduling performance centered on the Maximum Task Throughput metric.

II. Background and Terminology

A. Task Scheduling

In the context of this article, Task Scheduling involves the scheduling of elementary computational units called *tasks* to processor cores according to data dependency relationships between them. This paradigm resembles the out-of-order behavior of many modern processors, but allowing function calls (tasks), rather than instructions, to be automatically dispatched to different computational units [13].

Task dependencies are such that task B is said to depend on some task A if, and only if, B is generated after A and at least one of the following propositions is true:

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

Fig. 1 exemplifies how OpenMP 4.0 pragmas [14] can be used to spawn tasks. In that example, every time the outer loop is executed, one task encapsulating the `fun1` procedure is generated, while the inner loop generates tasks encapsulating `fun2`. The `fun3` call is not encapsulated by any task. The `IN` (read), `OUT` (write) and `INOUT` (read and write) constructs indicate how the tasks interact with their pointer parameters. Fig. 1 also contains the task graph for N = 3. Node labels reflect the order at which tasks are submitted.

In software, pointer-based dependency relationships can be evaluated in several ways. One solution relies on holding task-specific arrays of accessed pointers, so that one might easily check which pointers are touched by any particular task, and pointer-to-task hashmaps, for efficiently determining which task (or group of tasks) accesses any particular pointer. A hardware accelerator for doing the same might employ similar data-structures.

A detailed description of how dependency resolution is performed by the specific dependency-resolution accelerator employed in this work can be found in [15], while an account of how the Nanos6 Runtime solves the same problem in software can be found in [16].

B. Maximum Task Throughput (MTT)

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.

C. Internal Speedup

Measure of the average core utilization by task contents. In a system with N cores, core utilization will vary between 0 and N. It is closer to the maximum value whenever runtime overheads are negligible in comparison to task size and there is enough parallelism to maintain all cores occupied.

III. Proposed Architecture

A. Architecture Overview

Our work adds native Task Scheduling support to a Rocket Chip [17] processor by integrating it with the Picos Task Scheduling accelerator. This involves the introduction of two significant Chisel [18] modules: Picos Manager, which is instantiated once in the system and accessible to all cores, and the Picos Delegate module, instantiated once in each core. Fig. 2 provides an overview of the system.

Picos Delegate instances expose Task Scheduling capabilities to individual cores by implementing custom instructions. These instances interact with Picos through Picos Manager, which arbitrates the distribution of ready-to-run tasks to cores, ensures transaction atomicity, buffers Picos-CPU transactions to conceal downtimes, and conciliates the different queue interfaces used by Picos and other modules.

The TileLink module in the above figure is a system-wide bus synthesized automatically by the Rocket Chip generator, providing cache-coherent memory accesses to all connected agents. A Tile refers to a block consisting of a single core along with its accelerators and caches.

Further discussion of the nature and functionality of Rocket Chip, Picos, Picos Manager, and Picos Delegates can be found throughout the rest of this Section.

B. Rocket Chip

We take benefit of Rocket Chip to generate a 30-core RISCV processor with Linux support and cache parameters that maximize cache size within our FPGA resource constraints. We use its RoCC interface to define custom instructions that allow user-level programs to interact with the Picos HW task Scheduler, as we discuss in Subsection III-G.

Our FPGA prototype includes Rocket Chip instances with relatively large[1] private L1 caches (128 KB for data, 64 KB

[1]For comparison, the L1 data and instruction caches are, respectively, 4 and 2 times the size of those found in an AMD 7950x processor. Their large size aims to minimize, under the FPGA resource constraints, the need for higher-level caches.

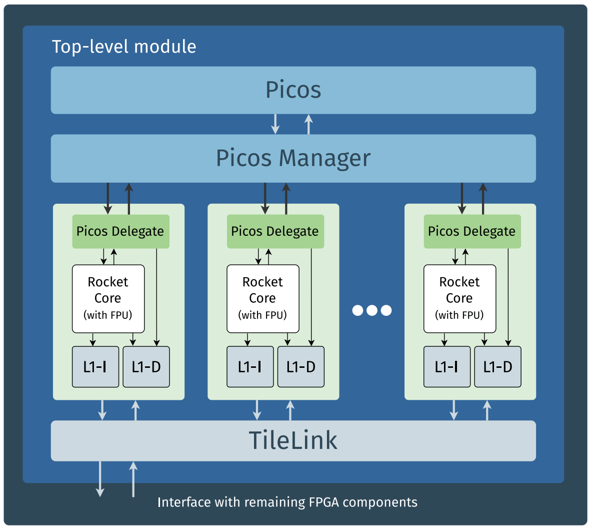


Fig. 2. Overview of the Picos + Rocket Chip system architecture.

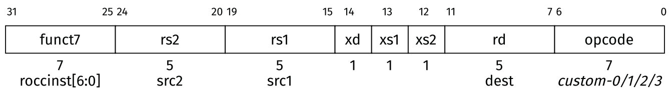


Fig. 3. Format of RoCC instructions.

for instructions) but no L2 caches, allowing us to fit many more cores than if a shared L2 cache was added. As a result, workloads issuing memory accesses with poor locality or exceeding L1 capacity should perform poorly in this system. In any case, this system characteristic makes it very capable to detect memory locality regressions that could be caused by the various evaluated Task Scheduling runtimes. Furthermore, since more realistic systems with shared L2 or L3 caches can perform inter-core communication in a much more efficient way, the scalability results we collect with our system can be aptly understood as lower bounds for what could be achieved by less constrained configurations.

C. RoCC Interface

This interface, present in all Rocket Chip cores by default, allows compliant accelerators to make cache-coherent memory accesses and be exposed to user programs through custom instructions. The RoCC instruction format is described by Fig. 3. 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.

A Rocket Chip Tile might include zero or more RoCC accelerators alongside its core and caches. In the system here described, all Tiles include a single instance of the Picos Delegate accelerator, which implements the task-related instructions described in Subsection III-J.

D. Picos

Picos is a Hardware Task Scheduling accelerator developed at the Barcelona Supercomputing Center supporting multiple worker classes (SMP cores, accelerators, etc) and nested task scheduling. In our system, Picos is configured to only feed SMP cores, and, for simplicity, all benchmarks are implemented in such a way that avoids nested tasks.

This accelerator has undergone a major revision since it was first integrated to a RISC-V system as reported in [12]. This revision changed its signal interface and packet API, requiring hardware modules handling Processor-Picos communication to be rewritten, but allowing scheduling overheads to be substantially reduced as a result of a more compact task packet representation.

E. Phentos

Phentos is a highly-optimized, light-weight, header-only C++ library that abstracts our custom Task Scheduling instructions, allowing for easier interaction with Task Scheduling software, and enabling tasks to be transparently distributed to workers from a fixed pool of threads.

While Phentos heavily relies on macros and inline functions for minimizing memory operations, its impact on instruction caches is very small compared to larger runtimes such as Nanos6 and OpenMP. In fact, compiling a Task Scheduling program with support for Phentos only impacts its binary size by less than 15 KB, while interaction with the shared libraries from those two non-accelerated runtimes requires several extra megabytes to be loaded to memory.

Phentos does not prevent context switches in any way. Also, to avoid deadlocks, Phentos allows task creation actions to be interleaved with task execution when submission is blocked, such as when Picos internal memory is full.

A mechanism must be in place to make sure that, after a task is submitted to Picos, the software runtime keeps track of its metadata (related function pointer, input parameters, etc) up to the point when Picos sends the task to a worker, which will require such metadata to execute the task. Picos could be configured to hold that information in memory, but doing so might considerably increase its on-chip resource utilization. Our integrated system thus implements two different software-based mechanisms for that, leading to two Phentos APIs (ORD-Phentos and FAST-Phentos), which only differ in their submission procedure, but not on other actions (such as work-fetching, task-waiting, signaling of task completion, etc). We shall detail their nature and tradeoffs in the following two Subsections.

1) ORD-Phentos: ORD-Phentos stores all task metadata on a custom-typed cache-aligned array such that each of its elements can hold a 64-bit task function pointer and either 7 or 15 input parameters. The 15-input version of this array takes 2 cache lines per element, doubling the requirements of the 7-input version, so the shorter version is used whenever the system does not include any task with more than 7 inputs, which is not rare, considering that constant scalar parameters might be held as global variables.

The 7-input configuration will thus generate one cache line write per submission, one cache miss per ready task fetched (not considering the loading of function instructions), and one cache line write for making the array entry as empty once the task finish executing. In total, for the 7-input configuration, 3 cache transactions are required for every task managed by the system, compared to 5 transactions for the 15-input configuration.

2) FAST-Phentos: The FAST-Phentos API was designed with the goal of substantially reducing the number of cache transactions required for managing task metadata, although restrictions apply to when it might be used, as shall be explained next. When a task application is amenable to it, FAST-Phentos might be used to reduce the number of metadata-related cache transactions per task by up to 100% when compared with ORD-Phentos, depending on the memory access patterns of the parallelized task kernels.

FAST-Phentos derives its benefits from two facts:

1) Task Parallel programs usually have very few different functions encoded as tasks.
2) Often, task inputs are of the kind `(base_pointer + constant * index)`, where the index can often be encoded with not more than 20 bits.

Observation (1) suggests that task function pointers might be stored in a global shared array, rather than being repeatedly propagated from the submission thread to worker threads for every task. Given that such tasks are very few, holding all their pointers on shared memory might not require more than one or two lines in the data caches from every core (up to 8 64-bit pointers might be held per 64 byte cache line). This allows function pointers, under certain conditions, to be directly fetched from L1 cache, rather than leading to a compulsory cache miss as with ORD-Phentos.

To understand and the significance of Observation (2), it is useful to acknowledge how ORD-Phentos identifies correspondences between ready-tasks made available by Picos and the metadata entries stored in processor memory during submission. This is achieved by simply having Picos refer to ready tasks using a 64-bit identifier provided by Phentos during task creation.

It is worth noting that the validity of Observation (2) depends solely on the workload being executed, not the processor bit-width. It is valid, for example, for workloads such that each of their task parameters is sampled from an array of no more than one million positions, where each of these array positions has some arbitrary *constant* size.

But as both Observations (1) and (2) indicate, using 64-bit identifiers for tasks (as defined by Picos API regardless of processor bit-width) is frequently very wasteful, as the number of combinations of task function pointers and input values is very limited. This allows FAST-Phentos to encode all the metadata from each task within 64-bits, in the form `(function_idx, input1_idx,..., inputn_idx)`[2]. Whenever this is not

[2]This is possible even if one of the fields takes more than the 20-bits suggested by Observation (2), provided that the remaining ones are small enough to still fit in a Picos-compliant 64-bit submission packet.

possible for all functions, Phentos-based applications might simply fallback to the ORD-Phentos API.

Compressing input values as indicated before might require additional shared variables to be kept in memory, such as when these inputs index some shared array. While that allows for a worst-case scenario where FAST-Phentos generates even more cache misses during work-fetches than ORD-Phentos (one for the function retrieval plus one per compressed input, compared to exactly one miss for 7-input ORD-Phentos), the fact that there are usually very few different memory regions indexed by these inputs frequently allows all of them to be kept within a single cache line. As a result, whenever task execution does not thrash all private data cache contents, work-fetching might recover both function address and input base pointers without incurring on any cache miss.

Furthermore, while ORD-Phentos requires, for each completed task, a memory write for marking its defunct metadata element as free to be overwritten, FAST-Phentos does not, given that it does not hold any task-specific data structure in memory.

Under the optimal scenario, FAST-Phentos eliminates all task handling cache misses, even though that is only possible when data touched by compute kernels fit in L1 cache. In the worst case, FAST-Phentos issues one cache miss per compressed task datum (input or function pointer).

Regardless of FAST-Phentos ability to hold arbitrary task metadata in cache for any given application, it never requires memory writes during task creation or termination, with favorable performance implications. RISC-V has a relaxed memory model, so inter-core data propagation requires explicit memory barriers that can impact unrelated memory operations, degrading performance. ORD-Phentos must store task metadata in a way that is visible to all worker cores, requiring such a barrier at the end of each submission and consequently limiting task creation rate. The same is not true for FAST-Phentos, so its task creation latency should be strictly lower than that of ORD-Phentos.

In summary, FAST-Phentos should display higher average performance than ORD-Phentos, even though performance degradation might be triggered in some cases.

F. The Nanos-RV Hardware Accelerated Runtime

Nanos-RV is a variant of the Nanos Task Scheduling Runtime where most SW-based dependency management code is replaced with calls to our HW-accelerated Task Scheduling instructions. Such use of hardware acceleration allows this variant to substantially outperform its non-accelerated baseline. At the same time, our experience with it suggested that its inherited Nanos complexity greatly impacted Task Scheduling throughput. Our work with this runtime thus motivated the design of a clean slate alternative, eventually materialized as Phentos. Given that the latter generally outperforms Nanos-RV by a large margin, the remaining of this work will mostly focus on Phentos.

G. 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 leverage the power of Picos to track task dependencies much faster than software runtimes but we also try 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.

TABLE I
CUSTOM TASK SCHEDULING INSTRUCTIONS SUPPORTED BY THE SYSTEM

Name	Description
Initiate Task	Informs the system about the swID and number of dependencies of a new task.
Add Info	Allows the runtime to inform Picos about task metadata relevant to nested task scheduling.
Send IN Dep	Used during task submission to encode a single 64-bit memory pointer referring to an IN dependency.
Send IN Deps	Used during task submission to encode two 64-bit memory pointers referring to IN dependencies.
Send OUT Dep	Used during task submission to encode a single 64-bit memory pointer referring to an OUT dependency.
Send OUT Deps	Used during task submission to encode two 64-bit memory pointers referring to OUT dependencies.
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.
Retire Task	Informs the system about the retirement of the task with the Picos ID given.
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.
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.

While designing Picos Manager and the auxiliary RoCC accelerator, we opted for making all the new instructions non-blocking. In this context, blocking instructions are those that only return after the corresponding transaction between Picos Manager and the core executing the instructions has completed. Making all instructions non-blocking gives more freedom for the runtime 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 is not able to service any of these requests, the 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 III-H.

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

H. Averting Deadlocks With 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, it might never recover from it.

This can happen because of the two following facts: (1) submissions fail when buffers and other data structures in 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 system stalls.

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 submit tasks to Picos. Let us further suppose that just prior to the execution of *Ready Task Request* by T, the centralized ready task request queue in the Work Fetch Controller[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 task descriptors in Picos, 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 all instructions (related to submission, work-fetching, and retirement) non-blocking, which allows a thread holding the responsibilities of both generating and running tasks to freely switch between these roles.

I. Avoiding Load Imbalance

Load imbalance refers to the uneven distribution of work among computation units (such as processor cores), often leading some of them to spend time idling, reducing average utilization rates and limiting maximum speedups with respect to serial execution. Our system avoids these problems by storing ready tasks in a single shared queue that all cores are allowed to fetch work from. Such work-pull operations are triggered by the *Ready Task Request* instructions described in Segment Section III.J.5.

Although the system allows for buffering of ready tasks by the cores, both our Nanos-RV and Phentos implementations avoid having multiple pending *Ready Task Request* operations, such that whenever such requests are fulfilled by Picos, the core receiving the new ready task immediately starts executing it. In this manner, the situation whereby a core keeps a ready task to itself while other cores starve for work is made impossible, and work stealing never becomes necessary. The core-private buffers thus behave as passthrough channels.

[3]This module, shown in Fig. 4, arbitrates ready task requests coming from all cores into a single routing queue, whose data determines the order at which requests are fulfilled.

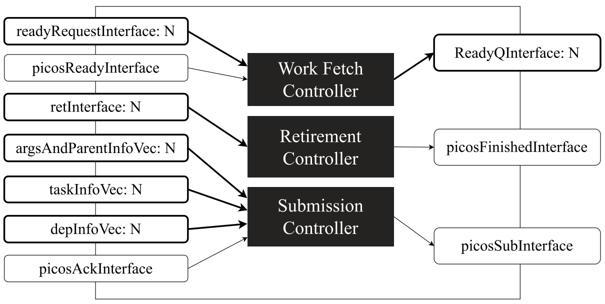


Fig. 4. Internals of the Picos Manager module.

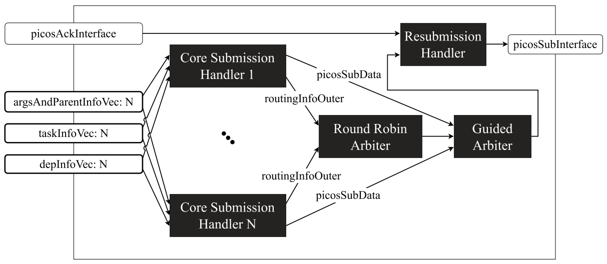


Fig. 5. Block diagram of the Submission Controller, a module instantiated by Picos Manager for carrying out transmission of new task descriptors to Picos.

J. The RoCC Accelerator

The ISA extension defined by our architecture is implemented by the RoCC-compliant Picos Delegate modules instantiated in every core, as described in what follows.

1) Initiate Task: The Picos Delegate RoCC accelerators implement this instruction by pushing swID and dependency count values to independent buffers implemented within the core-specific Submission Handler corresponding to the core executing this instruction (see Fig. 5). If both buffers can simultaneously accept the insertion, the instruction returns a success flag. Otherwise, a failure flag is issued.

2) Add Info: This instruction implements support for nested tasks by allowing tasks to be described as children of previous tasks. This is achieved by letting tasks be assigned a parentID.

This instruction is implemented in a way similar to that of *Initiate Task*: it leads the Picos Delegate handling the instruction to write parentID information to a buffer in the core-specific Submission Handler related to the core issuing the instruction. If the transaction succeeds, the instruction returns a success flag. If not, the failure flag is produced.

3) Send IN / OUT Deps: These two instructions are implemented by sending the two 56-bit pointers provided by the instruction caller, alongside a two-bit token indicating their IN, OUT, or invalid nature, to a core-specific buffer similar to those from *Initiate Task*. If that is possible, a success flag is returned. Otherwise, the failure flag is generated.

4) Send IN / OUT Dep: These are single-dep versions of the instructions in Section III.J.3.

5) Ready Task Request: Our RoCC accelerators 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. RoCC accelerators 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 work-fetch requests by any core after having satisfied R. Thus, Picos Manager distributes ready-to-run tasks in the same order that work-fetch requests come from the cores.

6) 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 RoCC accelerator instance in that core fulfills the instruction by returning a failure value; otherwise, it returns the SW ID encoded by the front element of the queue and setting an internal success flag. In either case, it does not pop q_i.

7) Fetch Picos ID: Suppose that core c_i, with private ready queue q_i, issues a *Fetch Picos ID* instruction. If q_i is empty, the RoCC accelerator instance in that core fulfills the instruction by returning a failure value; otherwise, 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.

8) Retire Task: The RoCC accelerator fulfills *Retire Task* instructions by pushing the payload of the operand register to the Retirement Controller in Picos Manager (see Fig. 4). If that operation might be completed within a single cycle, the instruction succeeds and a success flag is produced as a return value; otherwise, a failure flag is returned.

K. Picos Manager

Picos Manager arbitrates all data communication between Picos and individual cores. It serves as a protocol converter between the interface defined by core-specific Picos Delegates (which implement the custom RoCC instructions) and Picos itself. By virtue of that, in the event that the Picos interface is ever changed, only changes to Picos Manager are required, not to the cores.

1) Interface: As shown by Fig. 2, Picos Manager is connected to Picos and each of the core-specific RoCC accelerators (here called Picos Delegates). Its core-specific interface, which is replicated for each core, includes (1) a ready queue, (2) a retirement queue, (3) three submission queues, and (4) a work fetch request queue; its Picos-facing interface includes (5) a ready queue, (6) a retirement queue, and (7) a submission queue.

2) Structural elements: As described by Fig. 4, Picos Manager comprises three basic components: the Work-Fetch Controller, the Retirement Controller, and the Submission Controller. In the following lines, we will discuss the behavior and inner mechanics of each of them.

Submission Controller: This component — shown in detail by Fig. 5 — is the module that handles processing of submission packets in behalf of Picos Manager. It serves two 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) implement protocol crossing logic to ensure that communication between the various cores and Picos comply with Picos interface.

Picos Manager instantiates a Core Submission Handler for each core in the system. Each of these instances consumes data from the elementary submission queues coming from its corresponding core to build packet sequences compliant with Picos interface. Additionally, they interact with arbiters instantiated within the Submission Controller to secure permission for atomically sending data to Picos.

The `routingInfoOuter` interface from each Core Submission Handler contains a submission request describing the length of the corresponding submission sequence. The Guided Arbiter forwards data from the core whose submission request it receives through the Round Robin Arbiter, ensuring that packets from different submissions are never interleaved. The Round Robin Arbiter selects submission requests from the cores in round-robin fashion.

The Guided Arbiter does not send data directly to Picos, but to a Resubmission Handler, which allows submission actions to be re-attempted whenever Picos issues a negative acknowledgment signal indicating that it has not been able to handle the latest submission. That usually only occurs when internal Picos memories do not have space for additional in-flight tasks.

Each of the three elementary submission queues connected to each Core Submission Handler transmit data from a different class of submission instruction ({Initiate Task}, {Add Info}, or {Send IN Dep(s), Send OUT Dep(s)}).

Work-Fetch Controller: This module is responsible for distributing ready-to-run task descriptors to cores according to the total-order at which they requested such data.

Retirement Controller: This unit arbitrates retirement data coming from each core in the system. Collisions are frequent whenever core utilization is high and tasks are relatively small. When a collision occurs, this controller picks one core to send data in round-robin fashion and causes other cores to retry the retirement operation. This module is also responsible for converting single-packet retirement streams coming from the cores to three-packet retirement streams expected by Picos.

IV. Experimental Setup

A. System Characteristics

Each experiment is executed on a FPGA instantiation of the system described by Table II. As discussed in Subsection III-B, this system has several characteristics that make it very sensitive to excessive inter-core data traffic, such as having no shared caches, only one MSHR per core, as much as 30 cores, and employing a snoop-based coherence protocol rather than a directory-based one. As a result, the hardware-based Task Scheduling acceleration here described is likely to display even higher scalability in systems with more performant multi-core cache configurations.

The Linux 5.10.7 environment that all evaluated applications depend on is built using Buildroot 2021.8.1, which generates an initramfs (a memory-only file system) with the Linux kernel, system packages, and our benchmark binaries. The Linux kernel and basic packages are compiled from source by Buildroot, while the compilation of our binaries is handled separately. All ORD- and FAST-Phentos applications are built with RV-enabled GCC 10.3.0, while Nanos applications are compiled by Mercurium 2.3.0 [19], which transpiles application code into C and C++ temporary files that are finally compiled by GCC 10.3.0 as well.

All cores include a floating-point unit and custom RoCC instructions enabling interaction with Picos, being all symmetrical with respect to their HW Task Scheduling capabilities. Even so, to eliminate the effects of thread migration on application behavior, threads are locked to cores in all program executions in a way that cores $[1, N-1]$ are limited to task execution while core 0 is left to handle both task creation and execution, where N is the number of cores.

Internal speedups (average core utilization by task kernels rather than runtime overheads) are measured according to the following formula, where T is the set of all tasks of a program P, and $W(x)$ is the wall-time of x, in cycles:

$$S_i(P) = \frac{\sum_{t \in T} W(t)}{W(P)}$$

The wall-time of a task refers to the number of processor cycles elapsed during a task execution. It is measured by issuing *rdcycle* instructions immediately before and after the task payload (which is always a function) is called, to evaluate the difference between these cycle counts. All time-consuming operations are taken into account: cache misses, context switches, page faults, etc.

B. Benchmarks

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

1) The *blackscholes* application, from 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. It is a highly data-parallel application from Parsec [20].
2) The *sparseLU*, *jacobi*, *matmul*, and *dot-product* applications represent the Linear Algebra domain. The first of them solves pseudo-random sparse linear systems, the second uses the Jacobi iterative equation solver for solving the Poisson equation in one dimension, the third performs block-based matrix multiplication, and the last implements inner product calculation. Such programs are derived from the implementations found in the Kastors Benchmark Suite [21] and the `ompss-ee`[5] Github repository.
3) The *stream-deps* and the *stream-barr* programs are micro-benchmarks that evaluate system performance at handling routines of very high memory intensity. 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 found at the `ompss-ee` repository as well.
4) Finally, the *nbody* benchmark computes N-body gravitational interactions, representing the physics simulation domain.

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

TABLE II
Summary of System Characteristics

Number of Cores	30
Clock	60 MHz
Architecture	RV64G
Rocket-Chip version	Customized 525ddd37a
Front-end capabilities	In-order, single-issue
Number of MSHRs	1
L1 Data Cache size	128 KB
L1 Instruction Cache size	32 KB
L1 cache wayness	D-Cache: 16; I-Cache: 4
Cache line size	64 bytes
D-TLB topology	Fully associative, 32 entries
I-TLB topology	Fully associative, 32 entries
DDR capacity	16 GB
DDR generation	DDR4
DDR Clock	1200 MHz (2400 MT/s)
CAS latencies	Read: 17 cycles; write: 12 cycles
Number of memory channels	1
OS	Linux 5.10.7
Buildroot version	2021.8.1
GCC version	10.3.0
Mercurium version	2.3.0
Cache coherence protocol	MESI[4]

V. Results and Discussion

A. Comparing Phentos and Nanos

Fig. 6 summarizes how speedups over serial execution vary with respect to runtime and input selection. We can see that

[4]MESI implies a write-back policy and usage of snooping protocol.

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

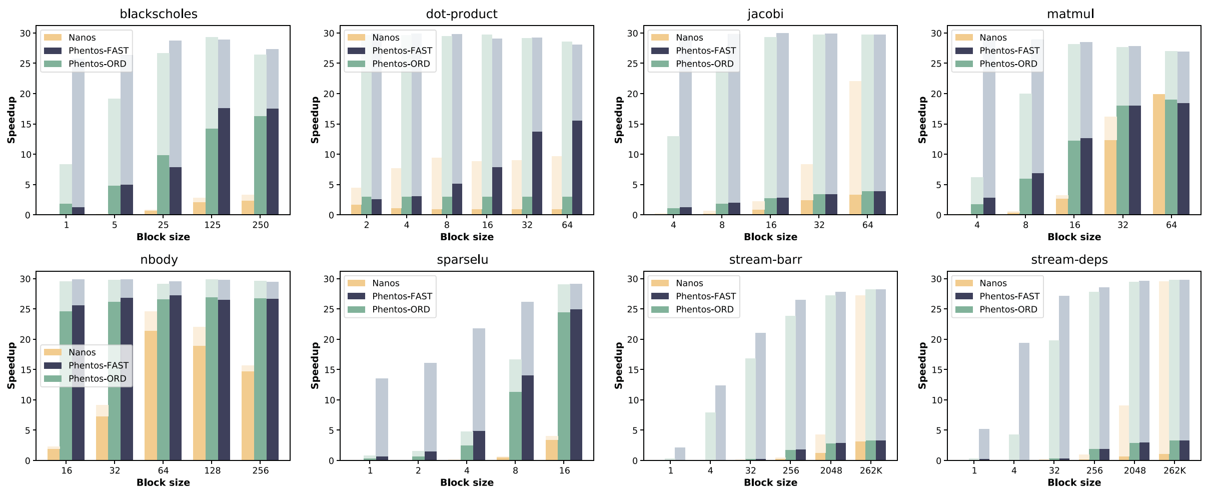


Fig. 6. Speedups at 30 cores for all benchmarks. Solid bars represent speedups over serial execution, muted bars show effective core utilization.

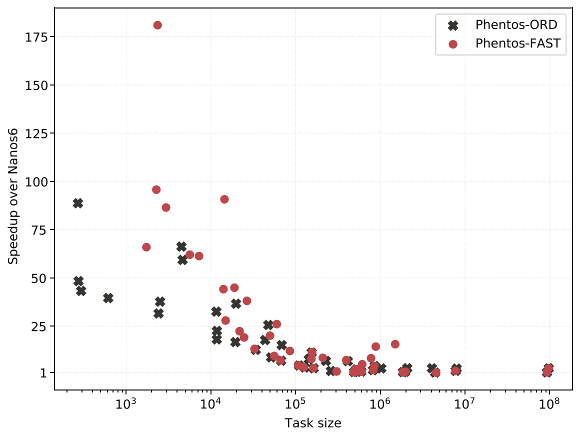


Fig. 7. Speedups of all applications executed on Phentos, taking benefit of HW-acceleration, over equivalent SW-only executions on Nanos6.

both Phentos variants outperform Nanos 41 out of 42 times, frequently by a substantial margin. The same figure also suggests that, as expected, such speedups are usually greater for larger block sizes. This is generally true up to the largest block size for which individual tasks do not exceed the data cache capacity of a single core. Experiments not displayed in this figure indicate that having tasks with larger work sets can lead to much poorer performance.

Fig. 7 summarizes the Phentos advantage with respect to Nanos, which it clearly suggests to be greatest for scenarios with small tasks. The geometric mean Phentos speedup over Nanos is around 7.5x for ORD-Phentos and 9.4x for FAST-Phentos. As expected, Phentos-over-Nanos speedups approach unity as task sizes increase, given that larger tasks more effectively amortize scheduling overheads and, provided that applications are sufficiently parallel, might saturate worker cores even if the Task Scheduling system is only capable of issuing a comparatively low amount of tasks per unit of time.

B. Deriving Theoretical Speedup Bounds From MTT

As described in Subsection II-B, Maximum Task Throughput (MTT) is the maximum number of tasks 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 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. Thus, one might derive a speedup bound MS for that system as a function of mean task size as the following:

$$MS(t) = min(N, K \times t)$$

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:

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

Having this in mind, for four different workloads, we measured the mean Task Scheduling overhead of Nanos-RV and Phentos, as shown by Fig. 8.

Fig. 8 clearly shows to which extent Nanos-RV and Phentos were able to reduce lifetime Task Scheduling overheads for varying workloads. In fact, Phentos presents lifetime overhead reductions of up to 253x with respect to Nanos-SW, while Nanos-RV shows reductions of up to 3.39x. Such measurements were taken with two different lifetime-overhead-measuring benchmarks: *Task Free*, which generates independent

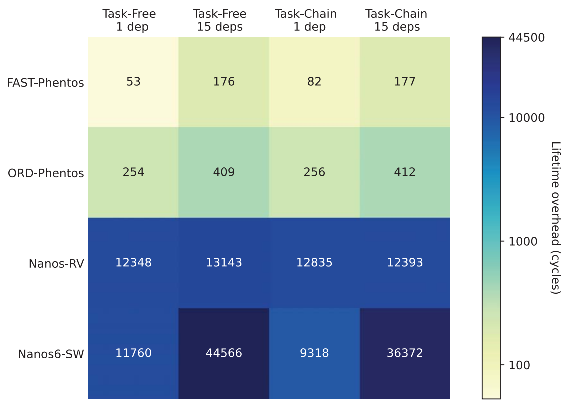


Fig. 8. Lifetime Task Scheduling overhead for several platforms, in Rocket Chip cycles.

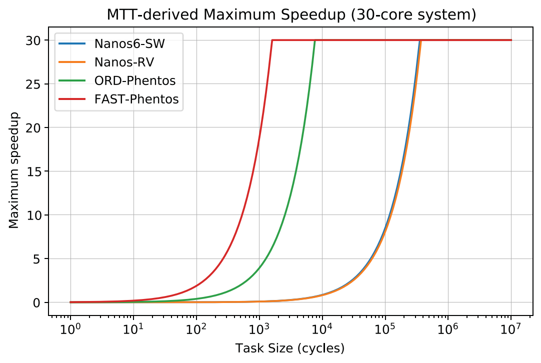


Fig. 9. Theoretical MTT-derived speedup bounds for several Task Scheduling platforms with thirty cores.

tasks with any number of monitored pointer parameters from 0 to 15; and *Task Chain*, which generates inter-dependent tasks forming a data dependency 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-Free (1 dep)* case and on Eq. (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 Fig. 9. 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. As an example, for task sizes around 10000 cycles, MTT-based maximum speedups for FAST- and ORD-Phentos are greater than 30x and 24x, respectively, while all other platforms have maximum speedups lower than 0.8x.

Finally, we overlay MTT upper bounds to performance data collected for each runtime on Fig. 10, where we can see that MTT curves serve as a strong performance limit for all runtimes, with no over-serial speedup or core utilization datapoint placed above it.

There, we can see that utilization figures are more likely to be close to MTT limits than over-serial speedups. This is mostly due to the fact that over-serial speedups can only exceed core utilization if the total computation time in the parallel scenario is smaller than the total computation time of a serial execution, which only occurs in the somewhat rare case where the parallel version is more cache amenable than the serial version. Among all reported datapoints, this only occurs for the `(Nanos, matmul, 64)` execution, where over-serial speedup slightly surpasses utilization. This workload benefits from the Nanos scheduling optimization that, given some core c retiring some task T, preferentially assigns tasks made ready by the completion of T to c, since that new task is likely to find relevant data produced by T in that core's cache. If this optimization is disabled, over-serial speedup drops by around 10% while utilization remains virtually the same.

For all considered benchmarks, both Phentos versions are generally capable of saturating cores with useful work (reaching an internal speedup close to 30) when block sizes are large enough, as suggested by Fig. 6. Nanos, on the other hand, can only approach doing so for half of the benchmarks, likely as a result of its lower MTT and its need to occupy worker cores with task management actions.

Moreover, since our internal speedup (effective utilization) measures exclude CPU runtime overheads, it tends to be smaller whenever these overheads take a substantial portion of CPU time. This is frequently the case for Nanos, since its non-accelerated nature requires all dependence management to consume CPU cycles both in the submission thread and the worker threads. This Nanos peculiarity is one reason why utilization is less likely to approach the MTT bound for this runtime than for either Phentos variant.

Still, while runtime overheads are generally much lower for Phentos than for Nanos, the task management overheads of both Phentos versions is still sensitive to the general memory behavior of the application being executed. This is because memory operations performed by either FAST- and ORD-Phentos to achieve data communication between the submission thread and the worker threads take different amounts of cycles to be completed depending on, among other things, average memory contention.

Finally, it is interesting to note that FAST-Phentos data in Fig. 10 seems to be, with respect to ORD-Phentos data, compressed beyond the 1K cycles vertical line. This follows from the fact that task sizes are also dependent on memory contention, given that the execution time of most tasks tends to be dominated by memory operations. Since FAST-Phentos tends to issue tasks to cores at a higher frequency than ORD-Phentos, tasks managed by FAST-Phentos tend to cause greater contention, which then cause these tasks to take more time to execute. In any case, this does not prevent FAST-Phentos to outperform ORD-Phentos in the general case or even in the instances where this effect is most noticeable, such as for the `(FAST-Phentos, sparselu, 1)` datapoint, where FAST-Phentos is able to outperform the other Phentos variant by more than 2x even with a much larger task size.

C. *Resource Utilization*

Table III showcases the resource utilization of several relevant system components. In particular, it shows that, for any given FPGA resource class, less than 3.8% of the

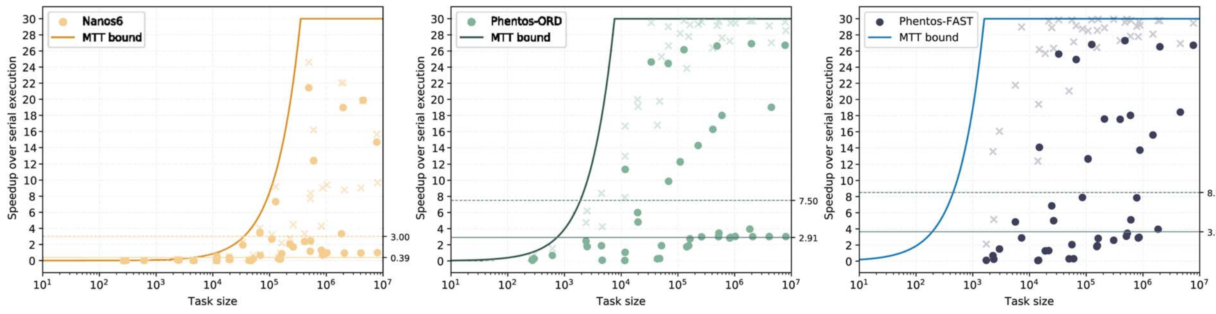


Fig. 10. Experimental speedup data of Task Scheduling applications over corresponding serial executions compared with theoretical MTT-derived bounds. Solid dots represent speedup over serial execution, cross markers depict internal speedups (effective core utilization). Dashed and solid horizontal lines represent arithmetic and geometric mean speedups, respectively. As it is the case for Fig. 9, MTT values are derived from Task-Free executions involving fifteen monitored pointer parameter par task.

TABLE III
Resource Usage Breakdown for Single Instances of Various Relevant System Modules, Including Submodules. Percentage Values for Any Resource Class Are Calculated With Respect to FPGA Capacity

Module	Cardinality	LUT	FF	BRAM
Alveo U-200	-	1182240	2364480	2160
Top level	One per system	89.8%	24.3%	93.9%
Core	30	2.65%	0.67%	2.73%
FPU	30 (one per core)	1.03%	0.16%	0.00%
D-Cache	30 (one per core)	0.44%	0.14%	2.22%
I-Cache	30 (one per core)	0.07%	0.04%	0.51%
Coherence Bus	One per system	1.32%	0.06%	0.00%
Delegate	30 (one per core)	0.04%	0.01%	0.00%
Picos Manager	One per system	1.49%	0.11%	0.00%
Picos	One per system	0.69%	0.54%	2.57%
Picos + Picos Manager + Delegates	One per system	3.34%	0.90%	2.57%

whole-design utilization of that resource is due to the Task Scheduling subsystem (comprising Picos, Picos Manager, and Delegates). Considering that the CPU cores are in-order, single-issue, and relatively simple, one expects 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. Moreover, the Task Scheduling subsystem has buffers that could be scaled down to further reduce resource utilization if needed.

VI. Limitations and Future Work

While our system significantly reduces Task Scheduling overheads with respect to solutions without hardware acceleration, much opportunity exists for further improving its memory behavior.

Ideally, tasks should be allocated to cores in a way that maximizes cache locality, avoiding unnecessary bus contention and cache misses. In its present form, our system significantly departs from that ideal by allocating tasks to cores in near random fashion, not making any effort to ensure better data reuse.

The negative impact of random work allocation is greater for systems with larger numbers of cores. This is partially explained by the fact that if N idle workers are available, the probability that any given task will be assigned to the optimal core is $\frac{1}{N}$, assuming that no cores have equal allocation fitness for that task. Consequently, random allocation impairs our system's scalability, and should be replaced with a more cache-sensitive allocation strategy in future revisions of our integration.

Concretely, we plan to develop a new configuration where cache-aware task clustering is performed prior to task allocation, such that tasks sharing substantial amounts of data belong to the same cluster and tasks from the same cluster are assigned, when possible, to the same worker, maximizing cache temporal locality. Strong theoretical and simulation-based arguments have been made in favor of such approach [22], [23], but its practical hardware implementation within a SMP system, with support for fine-grained tasks, remains to be achieved.

Task Scheduling systems hold precise information on the data dependencies among future tasks, and it should be possible to leverage that information to design improved cache replacement policies. Such a policy should be more likely to discard data that is not going to be used by the next task coming to a worker, and less likely to evict data that is certainly going to be needed by that next task. If hints are provided to the HW Task Scheduler about the size of the various memory regions accessed by each task, it could even prevent data that is not going to be re-accessed in the near future from being cached, making caches strongly resilient to scan and thrashing access patterns.

Other cache replacement policies exist that provide these resilience properties [24], [25], [26], [27], [28], but they usually act in a reactive way, learning memory access patterns after the fact, and performing poorly until enough data has been gathered about the current application phase. Task-aware policies should not suffer from this issue, given that they would be based on certain or highly-likely future application behavior, and hold promise to be especially advantageous for applications that frequently switch between very disparate program stages.

As described in Subsection IV-A, our system blocks thread migration. This is in place to reduce execution time variability and, more importantly, ensure CPU-Picos transactions are not interrupted by thread movements, possibly corrupting Picos submission packet sequences. On the other hand, the work in [29] describes the advantages of supporting *untied tasks*, that is, tasks that might resume execution from a different thread after being interrupted for any reason. Such flexibility might be beneficial to load balancing, while possibly reducing data locality and adding context migration overheads. Minimizing such overheads was one of the main contributions of [29]. Fig. 6 shows that our system already achieves good load balancing under the described test conditions. Yet, one could devise scenarios where adversarial software running on the same processor could impair the Task Scheduling application by excessively engaging one of the cores (the one holding the thread creating most tasks, for example). We could prevent such performance degradation by either adding full support for untied tasks or by lifting the requirement that Phentos threads do not move across cores. To ensure the integrity of CPU-Picos transactions is preserved, this requirement could be relaxed to ensure threads are fixed to cores during such communication but allowed to migrate at other times. This should be enough to reap some of the benefits of supporting untied tasks, but would still come short from allowing *work-first scheduling* as described in [29]. In the future, we plan to make a more thorough characterization of typical Task Scheduling workloads to assess whether the additional complexity required by full untied task support are justified by any improvements in load balancing it might uncover.

Picos is built in such a way that, if its task-nesting functionality is used, interfacing hardware and software are required to implement a certain deadlock-avoidance fallback mechanism. The exact scenario where this behavior must be triggered, as well as the workload classes that might produce it, are described in [30]. Whenever that condition occurs, the fallback mechanism will enforce new ready tasks to run on the same thread where they were created until the deadlock condition is averted. In the near future, we plan to build this functionality into Picos Manager and Phentos so that the system might exploit nested parallelism. No changes would be required to the proposed ISA extension, and the performance of applications not using nesting should not be affected [30].

Finally, we note that our current system cannot simultaneously handle more than one Picos-enabled application. This is inconvenient in several ways, and particularly in that it limits system throughput when each application execution does not have enough intrinsic parallelism to utilize all available workers. Moved by this, we plan to include basic virtualization support to new versions of the system, such that the HW Task Scheduler might simultaneously hold and process information from the disjoint virtual memory spaces of different applications. This would let the system handle not only context switches between a single Picos-based application and multiple non-Picos applications, but also between any number of Picos applications.

VII. Related Work

Our approach for supporting fine-grained Task Scheduling relies on minimizing the overhead for maintaining a dynamic task graph. The system proposed in [31] avoids maintaining such a data structure by allowing tasks to execute speculatively, taking benefit of Intel's Transactional Synchronization Extensions. Simulated results indicate that it should provide compelling performance, provided that transactions are fine-grained enough to avoid high abort rates. One downside of speculative systems such as this is that abort decisions might be based on conservative ordering constraints, possibly limiting parallelism [32]. Also, it might be difficult to simultaneously achieve fine-grained parallelism and low abort rates for some applications with many data dependencies per task [33].

Some works have been proposed in the past that also attempt to reduce Task Scheduling overheads with HW acceleration. Nevertheless, they come short in either not providing detailed full system evaluation, with FPGA prototyping or at least RTL simulation [13], [34], [35], [36]; only being able to feed tasks to a handful of general-purpose cores [30], or none at all [15]; or having their performance strongly limited by poor CPU-accelerator communication mediated by main memory [37], [30], rather than by custom datapaths and instructions as in our case.

Task Scheduling, as supported by our system, can be understood as a means to approximate dataflow behavior on multicore CPUs [13]. Special-purpose dataflow architectures find ample use in machine learning accelerators, and were proven useful since the first attempts to accelerate CNNs with fixed-function hardware [38], [39]. Some recent works have attempted to improve the programmability of CGRA-based dataflow systems by increasing their ability to handle complex control-flow [40]. Others propose processors with a tree-like microarchitecture that is specially apt at mapping irregular DAG applications [41]. A holistic Task Scheduling solution is presented in [42], where a HW task scheduler with the ability to drive CPUs, GPUs, and FPGAs is described. Other approaches use Task Scheduling program representations to automatically synthesize equivalent hardware [43] or configure dataflow systems [44]. These solutions offer substantial energy and latency advantages over ordinary CPU or GPU execution, but lack the versatility that these baselines or our proposal offer. Also, some of these works limit their evaluation to pre-RTL software simulation [42], [44].

VIII. Conclusion

This work presents a hardware-software co-designed architecture allowing for efficient Task Scheduling on large multicore systems. By enabling Phentos, a novel light-weight Task Scheduling runtime, to access HW-acceleration through low latency custom RoCC instructions, we reduce scheduling overheads by up to 253x, affording task parallel workloads with tasks as short as 10K cycles long to saturate the 30 cores of our FPGA-based RISC-V multiprocessor. We also point out the sensitivity of such a system to cache behavior, suggesting that new revisions of this architecture include cache-aware task placement mechanisms for reducing memory contention and miss frequency.

References

[1] Q. Chen, B. Tian, and M. Gao, "Fingers: Exploiting fine-grained parallelism in graph mining accelerators," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2022, pp. 43–55.

[2] "Pthreads (7): Linux manual page," Man7.org, Munich, Germany, 2021. [Online]. Available: https://man7.org/linux/man-pages/man7/pthreads.7.html

[3] "MPI: A message-passing interface standard version 4.0," Garching, Germany: Message Passing Interface Forum, Jun. 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[4] "CUDA C++ Programming Guide," NVIDIA Corp., Santa Clara, CA, USA, Tech. Rep., 2023. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[5] K. Berlin et al., "Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures," in *Proc. Int. Works Lang. Compilers Parallel Comput.* College Station, TX, USA: Springer, 2003, pp. 194–208.

[6] A. Stamatakis and M. Ott, "Exploiting fine-grained parallelism in the phylogenetic likelihood function with MPI, Pthreads, and openMP: A performance study," in *Proc. IAPR Int. Conf. Pattern Recognit. Bioinf*. Melbourne, Australia: Springer, 2008, pp. 424–435.

[7] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, 1998.

[8] "Intel oneAPI specification v1.2," Intel Corp., Santa Clara, CA, USA, 2022. [Online]. Available: https://spec.oneapi.io/versions/1.2-rev-1/

[9] A. Duran et al., "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Process. Lett.*, vol. 21, no. 2, pp. 173–193, 2011.

[10] Google LLC. *The TensorFlow Library*. (2023). TensorFlow. [Online]. Available: https://www.tensorflow.org/

[11] *The PyTorch Library*. (2023). *PyTorch*. [Online]. Available: https://pytorch.org/

[12] L. Morais et al., "Adding tightly-integrated task scheduling acceleration to a RISC-V multi-core processor," in *Proc. 52nd IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 861–872.

[13] Y. Etsion et al., "Task superscalar: An out-of-order task pipeline," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*. Piscataway, NJ, USA: IEEE, 2010, pp. 89–100.

[14] *OpenMP Application Program Interface Version 4.0*. (2018). OpenMP Architecture Review Board. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[15] J. M. de Haro et al., "OmpSs@FPGA framework for high performance FPGA computing," *IEEE Trans. Comput.*, vol. 70, no. 12, pp. 2029–2042, Dec. 2021.

[16] D. Álvarez, K. Sala, M. Maroñas, A. Roca, and V. Beltran, "Advanced synchronization techniques for task-based runtime systems," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 334–347.

[17] K. Asanović et al., "The rocket chip generator," EECS Department, University of California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-17, Apr. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[18] J. Bachrach et al., "Chisel: Constructing hardware in a Scala embedded language," in *Proc. DAC*, 2012, pp. 1216–1225.

[19] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos Mercurium: A research compiler for openMP," in *Proc. Eur. Works OpenMP*, 2004, vol. 8, p. 56.

[20] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proc. 5th Annu. Workshop Model., Benchmarking Simul.*, 2009, vol. 2011, p. 37.

[21] P. Virouleau et al., "Evaluation of openMP dependent tasks with the Kastors Benchmark Suite," in *Proc. Int. Workshop OpenMP*. Salvador, Brazil: Springer, 2014, pp. 16–29.

[22] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis, "Locality-aware task management for unstructured parallelism: A quantitative limit study," in *Proc. 25th Annu. ACM Symp. Parallelism Algorithms Archit.*, 2013, pp. 315–325.

[23] M. K. Bhatti et al., "Locality-aware task scheduling for homogeneous parallel computing systems," *Computing*, vol. 100, no. 6, pp. 557–595, 2018.

[24] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 60–71, Jun. 2010.

[25] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "Ship: Signature-based hit predictor for high performance caching," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2011, pp. 430–441.

[26] A. Jain and C. Lin, "Hawkeye: Leveraging Belady's algorithm for improved cache replacement," in *Proc. 2nd Cache Replacement Championship*, 2017, pp. 1–4.

[27] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 413–425.

[28] E. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *Proc. Int. Conf. Mach. Learn.* PMLR, 2020, pp. 6237–6247. [Online]. Available: https://arxiv.org/abs/2006.16239

[29] G. Tagliavini, D. Cesarini, and A. Marongiu, "Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight openMP tasking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 2150–2163, Sep. 2018.

[30] X. Tan, J. Bosch, C. Alvarez, D. Jiménez-González, E. Ayguadé, and M. Valero, "A hardware runtime for task-based programming models," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 9, pp. 1932–1946, Sep. 2019.

[31] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "Unlocking ordered parallelism with the swarm architecture," *IEEE Micro*, vol. 36, no. 3, pp. 105–117, May/Jun. 2016.

[32] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez, "Fractal: An execution model for fine-grain nested speculative parallelism," in *Proc. 44th Int. Symp. Comput. Archit.*, 2017, pp. 587–599.

[33] Á. R. Perez-Lopez, "Puppetmaster: A certified hardware architecture for task parallelism," Ph.D. dissertation, Cambridge, MA, USA: MIT, 2021.

[34] C. Wang, X. Li, J. Zhang, X. Zhou, and X. Nie, "MP-Tomasulo: A dependency-aware automatic parallel execution engine for sequential programs," *ACM Trans. Archit. Code Optim. (TACO)*, vol. 10, no. 2, p. 9, 2013.

[35] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 162–173, 2007.

[36] F. Yazdanpanah, C. Álvarez, D. Jiménez-González, R. M. Badia, and M. Valero, "Picos: A hardware runtime architecture support for OmpSs," *Future Gener. Comput. Syst.*, vol. 53, pp. 130–139, Dec. 2015.

[37] X. Tan, J. Bosch, M. Vidal, C. Álvarez, D. Jiménez-González, E. Ayguadé, and M. Valero, "General purpose task-dependence management hardware for task-based dataflow programming models," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*. Piscataway, NJ, USA: IEEE, 2017, pp. 244–253.

[38] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 247–257.

[39] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 367–379, 2016.

[40] G. Gobieski et al., "A programmable, energy-minimal dataflow compiler and architecture," in *Proc. 55th IEEE/ACM Int. Symp. Microarchitecture (MICRO)*. Piscataway, NJ, USA: IEEE, 2022, pp. 546–564.

[41] N. Shah, W. Meert, and M. Verhelst, "DPU-v2: Energy-efficient execution of irregular directed acyclic graphs," in *Proc. 55th IEEE/ACM Int. Symp. Microarchitecture*. Piscataway, NJ, USA: IEEE, 2022, pp. 1288–1307.

[42] J. M. M. Diaz, K. Harms, R. A. H. Guaitero, D. A. R. Perdomo, K. Kumaran, and G. R. Gao, "The supercodelet architecture," in *Proc. 1st Int. Workshop Extreme Heterogeneity Solutions*, 2022, pp. 1–6.

[43] Y. Chi, L. Guo, J. Lau, Y.-k. Choi, J. Wang, and J. Cong, "Extending high-level synthesis for task-parallel programs," in *Proc. IEEE 29th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*. Piscataway, NJ, USA: IEEE, 2021, pp. 204–213.

[44] V. Dadu and T. Nowatzki, "Taskstream: Accelerating task-parallel workloads by recovering program structure," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2022, pp. 1–13.

Lucas Morais received the degree in computer engineering from the University of Campinas, Brazil, being accoladed as the best student of the class by the Engineering Council of the São Paulo State, and the M.Sc. degree in computer science from the University of São Paulo, where he worked on adding task parallelism support to RISC-V SMP systems. He is currently working toward the Ph.D. degree with Universitat Politècnica de Catalunya (UPC), focusing on HW-accelerated task parallel programming models, in concert with his work as a Research Engineer at the Barcelona Supercomputing Center.

Carlos Álvarez received the M.S. and Ph.D. degrees in computer science from UPC, in 1998 and 2007, respectively. He is currently a Tenured Assistant Professor with the Department of Computer Architecture, UPC, and as an Associated Researcher with Barcelona Supercomputing Center (BSC). His research interests include parallel architectures, runtime systems, and reconfigurable solutions for high-performance multiprocessor systems.

Daniel Jiménez-González received the M.S. and Ph.D. degrees in computer science from UPC, in 1997 and 2004, respectively. He holds a position as a Tenured Assistant Professor with the Department of Computer Architecture, UPC, and as an Associated Researcher with BSC. His research interests include parallel architectures, runtime systems, and reconfigurable solutions for high-performance computing systems to improve HPC and bioinformatic applications.

Juan Miguel de Haro is working toward the Ph.D. degree with the Computer Architecture Department, UPC. He works with the OmpSs@FPGA team, and he is involved in the Designing RISC-V-based Accelerators for next generation Computers (DRAC) project at BSC. Within OmpSs@FPGA, his work is focused on the hardware runtimes and accelerators implemented on the FPGA, as well as the communication between FPGAs.

Guido Araujo received the Ph.D. degree in electrical engineering from Princeton University, in 1997. He is a Full Professor in computer science and engineering with UNICAMP. His current research interests include code optimization, parallelizing compilers, and compiling for accelerators, in close cooperation with industry partners. He has published over 120 scientific papers and received five best paper awards. He was awarded two Inventor of the Year and two Zeferino Vaz Research Awards from UNICAMP. His students received two Best Ph.D. Thesis Awards from the Brazilian Computing Society. He co-founded the companies Kryptus and Idea! He is currently a member of the Editorial Board of the IEEE MICRO.

Michael Frank (Senior Member, IEEE) received the degree in physics from Technische Universität Darmstadt, in 1986. He is currently the Vice President and Chief Architect with Arteris IP. He previously served as a Senior or Principal hardware architect with companies such as Apple, where he worked on the A4 processor, AMD, ATI, and LG Electronics. Since 1989, he has been focusing on the microelectronics field, when he founded Xtended Design, a HW/SW design and consulting company.

Alfredo Goldman (Senior Member, IEEE) received the Ph.D. degree from INPG, Grenoble. He is an Associate Professor with USP, and a Subject Area Editor in parallel computing. He was the Co-Program Chair of SEMISH and the LATAM School in software engineering, in 2022. He was the Track Chair of EuroPar 2017 and 2021, SCC 2020, and CARLA 2022. His research interests include parallel/distributed computing, scheduling, and agile methods. He is on the board of governors of the Brazilian Computer Society and a member of ACM.

Xavier Martorell (Member, IEEE) received the M.S. and Ph.D. degrees in computer science from the UPC, in 1991 and 1999, respectively. Since 2001, he has been an Associate Professor with the Computer Architecture Department at UPC. His research interests include cover operating systems, runtime systems, compilers, and applications for high-performance multiprocessor systems. Since 2005, he has been the Manager of the Parallel Programming Models team at BSC. He has participated on several EU projects related to the use of FPGAs for HPC: AXIOM, EuroEXA, LEGaTO, and now Textarossa and MEEP. He has co-authored more than 80 publications in international journals and conferences.