Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale

WP2 New accelerator designs exploiting mixed precision

D2.11 IP for fast task scheduling, part 2

This project has received funding from the European Union's Horizon 2020 research and innovation programme, EuroHPC JU, grant agreement No 956831

TEXTAROSSA

Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale

Grant Agreement No.: 956831

Deliverable: D2.11 IP for fast task scheduling, part 2

Project Start Date: 01/04/2021 **Duration**: 36 months

Coordinator: *AGENZIA NAZIONALE PER LE NUOVE TECNOLOGIE, L'ENERGIA E LO SVILUPPO ECONOMICO SOSTENIBILE – ENEA, Italy.*

Deliverable No	D2.11
WP No:	WP2
WP Leader:	CINI-UNIPI
Due date:	M30
Delivery date:	30/11/2023

Dissemination Level:

PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

This project has received funding from the European Union's Horizon 2020 research and innovation programme, EuroHPC JU, grant agreement No 956831

DOCUMENT SUMMARY INFORMATION

Project title:	Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale
Short project name:	TEXTAROSSA
Project No:	956831
Call Identifier:	H2020-JTI-EuroHPC-2019-1
Unit:	EuroHPC
Type of Action:	EuroHPC - Research and Innovation Action (RIA)
Start date of the project:	01/04/2021
Duration of the project:	36 months
Project website:	textarossa.eu

WP 2 New accelerator designs exploiting mixed precision

Deliverable number:	D2.11					
Deliverable title:	IP for fast task scheduling, part 2					
Due date:	M30					
Actual submission date:	02/12/2023					
Editor:	Carlos Álvarez					
Authors:	A. Filgueras, M. Vidal, C. Alvarez, D. Jimenez, X. Martorell, L. Morais, JM. deHaro					
Work package:	WP2					
Dissemination Level:	Public					
No. pages:	45					
Authorized (date):	30/11/2023					
Responsible person:	Carlos Álvarez					
Status:	Plan	Draft	Working	Final	[✓ Submitted]	Approved

Revision history:

Version	Date	Author	Comment
0.1	2023-10-28	A. Filgueras	Draft structure
0.2	2023-11-02	C. Alvarez	New sections
0.3	2023-11-02	L. Morais	RISC-V integration

Quality Control:

Checking process	Who	Date
Checked by internal reviewer	F. Simula – O. Frezza	2023-11-28
Checked by Task Leader	Carlos Alvarez	2023-11-29
Checked by WP Leader	Sergio Saponara	2023-11-30
Checked by Project Coordinator	Massimo Celino	2023-12-01

COPYRIGHT

ACKNOWLEDGEMENTS

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement no 956831. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Italy, Germany, France, Spain, Poland.

Please see http://textarossa.eu for more information on the TEXTAROSSA project.

The partners in the project are AGENZIA NAZIONALE PER LE NUOVE TECNOLOGIE, L'ENERGIA E LO SVILUPPO ECONOMICO SOSTENIBILE (ENEA), FRAUNHOFER GESELLSCHAFT ZUR FOERDERUNG DER ANGEWANDTEN FORSCHUNG E.V. (FHG), CONSORZIO INTERUNIVERSITARIO NAZIONALE PER L'INFORMATICA (CINI), INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE (INRIA), BULL SAS (BULL), E4 COMPUTER ENGINEERING SPA (E4), BARCELONA SUPERCOMPUTING CENTER-CENTRO NACIONAL DE SUPERCOMPUTACION (BSC), INSTYTUT CHEMII BIOORGANICZNEJ POLSKIEJ AKADEMII NAUK (PSNC), ISTITUTO NAZIONALE DI FISICA NUCLEARE (INFN), CONSIGLIO NAZIONALE DELLE RICERCHE (CNR), IN QUATTRO SRL (in4). Linked third parties of CINI are POLITECNICO DI MILANO (CINI-POLIMI), Università di Torino (CINI-UNITO) and Università di Pisa (CINI-UNIPI); linked third party of INRIA is Université de Bordeaux; in-kind third party of ENEA is Consorzio CINECA (CINECA); in-kind third party of BSC is Universitat Politècnica de Catalunya (UPC).

The content of this document is the result of extensive discussions within the TEXTAROSSA © Consortium as a whole.

DISCLAIMER

Table of contents

List of Figures

List of Tables

List of Acronyms

Acronym	Definition
ASIC	Application Specific Integrated Circuit
BP	Business Plan
BRAM	Block Random Access Memory (in-FPGA RAM module)
CI/CD	Continuous Integration / Continuous Delivery
DSP	Digital Signal Processor
FF	Flip Flop
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FTS	Fast Task Scheduler
HPC	High-Performance-Computing
IP	Intellectual Property
ISA	Instruction Set Architecture
LLVM IR	Low-Level Virtual Machine Intermediate Representation
LUT	Look-Up Table
MMIO	Memory-mapped I/O
MPI	Message Passing Interface
PCIe	Peripheral Component Interconnect express
SMP	Symmetric Multi-Processor
URAM	Ultra Random Access Memory (in-FPGA large RAM module)

Executive Summary

This document reports on the activities done by TextaRossa partner BSC with reference to the design of the Fast Task Scheduler IP in WP2 and preliminary design and synthesis results, mainly in FPGA technology.

This deliverable is the second part of Deliverable 2.10 “IP for fast task scheduling, part 1”. As a demonstrator deliverable, Deliverable 2.10 contains the IP core design, including its workflow, interface and operating modes. Deliverable 2.10 also includes the IP algorithmic implementation details, the IP verification & validation process and the FPGA resource usage of the IP. After presenting the performance results of the IP, Deliverable 2.10 also includes information on how to obtain the source code of the IP developed, both as standalone design and as part of the larger OmpSs@FPGA framework. Finally, Deliverable 2.10 includes a copy of the description of the command software interface of the IP. After a review, the final Deliverable 2.10 was delivered in month 25 (only 7 months ago) and included the final revised version of the Fast Task Scheduling IP.

Consequently, this deliverable does not include the same information as deliverable 2.10 but adds the evaluation and integration of the developed Fast Task Scheduling IP in a many-core RISC-V environment that also uses BSC dependence manager IP. The IP has been further thoroughly tested through the project as reported in Deliverables D4.6, D4.7 and D4.8. The results reported in this deliverable have been published as a research paper in the Transactions on Computers journal.

1 Introduction

The main objective of developing an IP for fast task scheduling is to provide an effective and efficient way to send tasks to accelerators implemented in the FPGA. The scheduler IP allows to offload the process of scheduling tasks into individual accelerators and keeping track of accelerator status and finished tasks. This reduces the communications and synchronizations between host and FPGA accelerators, increasing overall performance. The main standalone evaluation of the IP is reported in previous Deliverable 2.10 "IP for fast task scheduling, part 1".

We envision two main fields of application for the IP developed in this deliverable. The first one is as part of the OmpSs@FPGA framework where this IP will be an integral piece of the hardware runtime (in fact it will be the hardware runtime itself). In the OmpSs@FPGA framework application the Fast Task Scheduler is expected to communicate with the host CPU fast enough that it does not represent a bottleneck to the scheduling of small tasks to FPGA accelerators as it happens with software schedulers. This will allow task-based designs to be competitive and even outperform streaming applications by improving the shared use of the FPGA resources. This application scenarios are reported in deliverables D4.6, D4.7 and D4.8 and consequently not included here.

The second application scenario of the Fast Task Scheduling is the interconnection of different CPU cores and/or accelerators improving the performance of task-based programming models (like OpenMP or OmpSs). It has been demonstrated [3][4] that the runtime overhead can be the main bottleneck in the performance of manycore systems as the number of tasks should increase with the number of cores to take advantage of large systems. For this kind of problem, we aim to integrate the Fast Task Scheduling IP with a RISC-V manycore system and demonstrate significant performance improvements. This work is reported in this deliverable.

1.1 Relationship with project objectives

This deliverable is related to the following project objectives and strategic goals as stated in the DoA:

- Objective 1 - Energy efficiency. The IP reported in this deliverable is designed to be integrated in an FPGA or attached as a runtime accelerator to a manycore system. It provides two ways of increasing energy efficiency: a first-order effect by improving the energy efficiency of the task-based runtime and, a second-order effect that is achieved by improving the efficiency of the application being executed using the runtime. This effect is reported in this deliverable and in deliverables D4.6, D4.7 and D4.8 that also used the Fast Task Scheduler IP.
- Objective 2 - Sustained application performance. As with Objective 1, the IP reported in this deliverable contributes to sustained application performance: by improving the performance of the task-based runtime and also, by improving the performance of the application being executed using the runtime. As a fast task scheduling effectively increases application available parallelism, this second effect improvement is significant as reported in this deliverable and in deliverables D4.6, D4.7 and D4.8 that also used the Fast Task Scheduler IP.
- Objective 3 - Fine-tuned thermal policies integrated with an innovative cooling technology. The Fast Task Scheduling IP is expected to be able to work with the software part of the runtime by either, providing it with information about the power consumption of the tasks and/or enabling the thermal control system (in software) to actuate over the accelerators if necessary. This has been achieved as is reported in deliverable D4.6 "Task-based runtime systems".

- Objective 4 - Seamless integration of reconfigurable accelerators. OmpSs@FPGA runtime allows for seamless integration of reconfigurable accelerators (as detailed in deliverable D4.6, D4.7 and D4.8). As an integral part of the framework the IP should allow for scheduling of tasks that are either specific to an accelerator or destined to be executed in a general-purpose unit.
- Objective 5 - Development of new IPs. The first part of this deliverable, deliverable D2.10 reports the development of a new IP dedicate to scheduling tasks, so it directly tackles objective 5.
- Objective 6 - Integrated Development Platform. As part of the OmpSs@FPGA runtime the IP reported in this deliverable will be used in applications executing on the IDV-E platform. It is important to highlight that IDV-E features a host CPU (ARM based) that has never before been used to drive computation in a PCIe attached FPGA. Developing the system in a way that is compatible with new different CPUs helps ensuring new host CPUs (like EPI CPUs) will be able to drive this kind of computations in the future.

- Strategic Goal #1: Alignment with the European Processor Initiative (EPI). The Fast Task Scheduling IP will be aligned with EPI in both its application fields. On one side, along with OmpSs@FPGA, it will provide a system that can use an EPI processor to drive computations in a cluster of FPGA PCIe attached accelerators. Also, the second field of application of the Fast Task Scheduler is to accelerate a manycore system. This deliverable reports how to integrate the IP with a manycore system to accelerate parallel computations in such systems. The results have been published in a journal paper.
- Strategic Goal #2: Supporting the objectives of EuroHPC as reported in ETP4HPC's Strategic Research Agenda (SRA) for open HW and SW architecture. The Fast Task Scheduler IP is developed following the open HW model and is freely available as a standalone IP or as part of the whole OmpSs@FPGA framework.
- Strategic Goal #3: Opening of new usage domains. The IP reported in this deliverable addresses a problem that affects current manycore platforms: their inability to exploit large-scale parallelism when each of the pieces of work involved (tasks) is small. In this sense, although the IP itself does not address any specific usage domain removing this system bottleneck opens the possibility of executing efficiently new applications on the objective platforms.

1.2 Improvements over the state-of-the-art

The Fast Task Scheduler IP developed in TextaRossa and presented in this deliverable improves dynamic task scheduling by offloading scheduling work from the software to the hardware.

As commented before, one of the uses of the IP is its integration into the OmpSs@FPGA framework. OmpSs@FPGA uses the IP to schedule tasks to FPGA accelerators in order to improve efficiency and programmability. Other efforts try to improve the efficiency and programmability of FPGAs from High-Level Languages (HLL). The Vineyard project [9] aims at facilitating heterogeneous programming from OpenSPL [10], OpenCL [11] and SD-SoC [12]. The Ecoscale project [13] proposes a hybrid MPI+OpenCL programming environment and a minimum runtime system to orchestrate a large group of workers using reconfigurable accelerators. In both cases the approach is similar to the OmpSs@FPGA [1] baseline used in this work aiming at an easy usage of FPGA-based execution units. However, our approach also improves the overall system performance with high-level data access optimizations, task-based parallel execution and includes a complete hardware runtime in the FPGA fabric that also helps programmability. The Unilogic system [14] also proposes a small runtime to coordinate several FPGA accelerators at the same time but it is based on low-level code, reducing programmability. Mbongue et. al. [15] introduce automatic kernel extraction directly from LLVM IR code and use RapidWright [16], a tool that improves the placement and routing of the FPGA design by pre-compiling and replicating the kernels. As future work, the same idea could be applied to the OmpSs@FPGA accelerators potentially increasing the operating frequency of the whole design.

Several frameworks target High-Level Synthesis from C/C++. Vivado HLS [17] is the Xilinx tool that is used by OmpSs@FPGA to generate FPGA IP blocks. Xilinx Vitis [18] works on top of Vivado HLS to better integrate the execution environment with AMD-Xilinx boards. It is an evolution of Xilinx SDSoC [12] and SDAccel [19] environments that includes a minimum runtime to manage communication between the FPGAs and the SMP host and facilitates the use of several already programmed FPGA library functions. In the same direction, Intel oneAPI [20] and Quartus [21] allow the use of HLL for Intel FPGAs. LegUp [22], [23] is another HLS tool that synthetizes C code with Pthreads and limited OpenMP annotations. Each thread (code) is synthesized as an accelerator at compile time. The remaining (sequential) portions are executed in the processor, invoke accelerators and use synchronization functions to retrieve their return values. However, this framework only targets Microchip FPGAs [24]. ROCC [25], [26] was another interesting HLL compiler tool that was agnostic of the FPGA target. Our system is designed to be able to work over any HLS tool focusing on improving data movements and parallel execution of several accelerators, not the accelerators themselves. It also features a unique hardware runtime that allows more complex algorithms to be executed in the FPGA while improving performance. There have been some works related to hardware task management and using tasks to program FPGAs also. Tan et al. [3] present a HW manager that supports task dependencies resolution and heterogeneous task scheduling for parallel task-based programming models. However, the proposal only allows task offloading to the accelerators and was not integrated with any compiler framework. Cabrera et al. [27] and Sommer et al. [28] propose extensions in OpenMP to support the definition of tasks that target an FPGA device. Bosch et al. [2] also proposed to create tasks from inside the FPGA. However, these works were not developed to be integrated with a hardware dependence management system to allow fast task management inside the FPGAs.

Regarding the second use, the integration of a Task Scheduler IP into a many-core system, the most similar approach is the development of HW-acceleration strategies to reduce the overhead for dynamically detecting inter-task data dependencies, distributing tasks to workers according to such dependencies, and reacting to work completion events. Task Superscalar [4], MP-Tomasulo [5], Carbon [6], Nexus#[7], and Picos [8], [3] are examples of such solutions. However, these works were limited by the fact that they either relied solely on high-level simulation, which makes it possible to neglect performance-critical considerations that arise from full-fledged HW/SW implementations, or, if they included an FPGA prototype, they did not support scheduling of software workloads to general-purpose cores rather than only fixed-function accelerators. Also, the integration of the dependence manager and the task scheduler did not allow the separate improvement of the different system components. Finally, some previous work analyzed the behavior of a preliminary hardware task scheduler with regular cores [37], but it only analyzed a small system with 8 cores using a feature limited system.

To the best of our knowledge, this work is the only one that allows dynamic task scheduling from an IP used by a high-level runtime developed for FPGA and ASIC integration with RISC-V CPU processors. In addition, it is the only one that allows task scheduling from a pragma annotated high-level source code that is automatically compiled into the final executable.

1.3 Document organization

This document describes both the high-level design and the inner-most functionality description of such IP. The document is organized as follows:

- Section 2 presents the motivation for IP core integration.
- Section 3 shows the design of the high-level architecture.
- Section 4 presents the instruction design.
- Section 5 explains the FTS manager module.
- Section 6 highlights the Phentos runtime.
- Section 7 details the experimental setup.
- Section 8 shows some significant performance results.
- Section 9 analyzes some theoretical results on the limits of the design.
- Section 10 concludes the report.

As a demonstrator deliverable, appendix A shows how to obtain the source code of the IP developed, both as standalone design and as part of the larger OmpSs@FPGA framework so it can be used without further development. Finally, Appendix B includes a copy of the description of the command software interface of the IP so a technology take-up of the IP could be undertaken. Finally, Appendix C shows a summary of the design of the IP available as a fast reference for the interested reader.

2 Multi-core RISC-V integration overview and motivation

As noted in the introduction, producing a tight-integration between the Fast Task Scheduler and a many-core CPU might be useful in several ways. First, in that communication latencies can be kept low by creating custom datapaths for leveraging scheduling functionality from the various processor cores. Secondly, in that this low overhead allows cores to be fed work-at a sufficiently high rate that their average occupation rate might be kept close to 100% even when only very small tasks are available. In contrast, a more weakly-coupled system could have communication overheads so high that the maximum rate at which they can feed the cores with work cannot keep up with the aggregate rate at which all cores would be completing tasks, leading to sub-optimal core utilization. Furthermore, these effects are even more dramatic for software-only systems not relying on any Task Scheduling acceleration. Our approach avoids these problems and was experimentally proven to deliver near perfect core utilization for a much ampler set of workloads than any competing system.

We use a 30-core Rocket Chip RISC-V system as our integration platform. The decision to use this environment was mainly due to its following characteristics:

- Well-defined HW interface for extending its ISA with custom instructions, allowing for the easy expression of CPU-Scheduler interactions as processor instructions.
- Relative low resource utilization, allowing for the instantiation of large numbers of cores (30+ in an U200 Xilinx FPGA).
- Its highly customizable nature, allowing for different numbers of cores, cache configurations, cache replacement policies, and FPU instantiation strategies to be seamlessly tried.
- The maturity and dependability of the IP, avoiding issues to execute long running and complex workloads.

In particular, we emphasize that relying on the development of custom instructions for handling CPU-Scheduler interactions was instrumental to avoiding typical latency-increasing communication pitfalls. For example, the alternative approach of kernel-level MMIO would trigger the use of system calls that add hundreds of cycles of latency to every interaction. Throughout the lifecycle of every task, that could lead to the accumulation of thousands of cycles of latency, nearly erasing all benefits of employing HW-accelerated TS in the first place.

Additionally, employing custom instructions has advantages even over user-level MMIO – which avoids the costly kernel system calls. Custom instructions allow for data to be processed before being sent to the accelerator, reducing the accelerator complexity. Beyond mere data forwarding, this processing can leverage a rich view of the current processor state before sending data to the accelerator, something that would either be more inefficient or impossible to achieve with an equivalent SW-based request preparation. Finally, custom instructions tend to more clearly express programmer intent than data-preparation macros and masks driving user-level MMIO interactions, where manual bit manipulations can obfuscate subtle software issues.

Given that our research aims at accelerating HPC applications, it was essential that our base platform included support for a multi-threaded Linux environment – not being limited to bare-metal execution – and HW-based floating point arithmetic. Both targets are met by Rocket Chip, which allows for the generation of FPU-enabled cores and that support a fully functional multiprocessor Linux environment with pthreads, OpenMP, Nanos, and other parallel processing runtimes.

As previously noted, care must be taken not to negate the benefits of HW acceleration with communication or processing latencies occurring at other system layers. Apart from optimizing HW-Scheduler datapaths, this requires careful design of the software libraries – effectively user-level drivers – that make the Scheduler capabilities available to the various relevant software workloads. More specifically, given that the HW Scheduler will typically not take more than 50-300 cycles to process a task throughout its whole lifetime, it is highly desirable that each task processing step performed in software – loading of task metadata, task creation, etc. - takes less than 100 cycles.

To achieve this, we developed a lightweight, high-performance runtime called Phentos, that provides full access to the Scheduler functionality as exposed by the custom instructions while avoiding the overheads of more complex runtimes such as Nanos or OpenMP. Such runtimes have a much larger codebase and need to balance performance with the need to implement features that are not relevant to our use cases, to the extent that it would be much more difficult to modify them to achieve our performance targets than to build a new minimal runtime such as Phentos from scratch.

As a result, we produced a system that strongly outperforms our SW-only Task Scheduling baseline, leading to end-to-end application speedups of up to 175x and to task-lifetime scheduling overhead reductions of up to 300x.

3 High-level architecture

Our work adds native Task Scheduling support to a Rocket Chip processor by integrating it with the Picos Task Dependence accelerator. This involves the introduction of two significant Chisel language modules: FTS Manager, which is instantiated once in the system and accessible to all cores, and the FTS Delegate module, instantiated once in each core. Figure 3.1 provides an overview of the system.

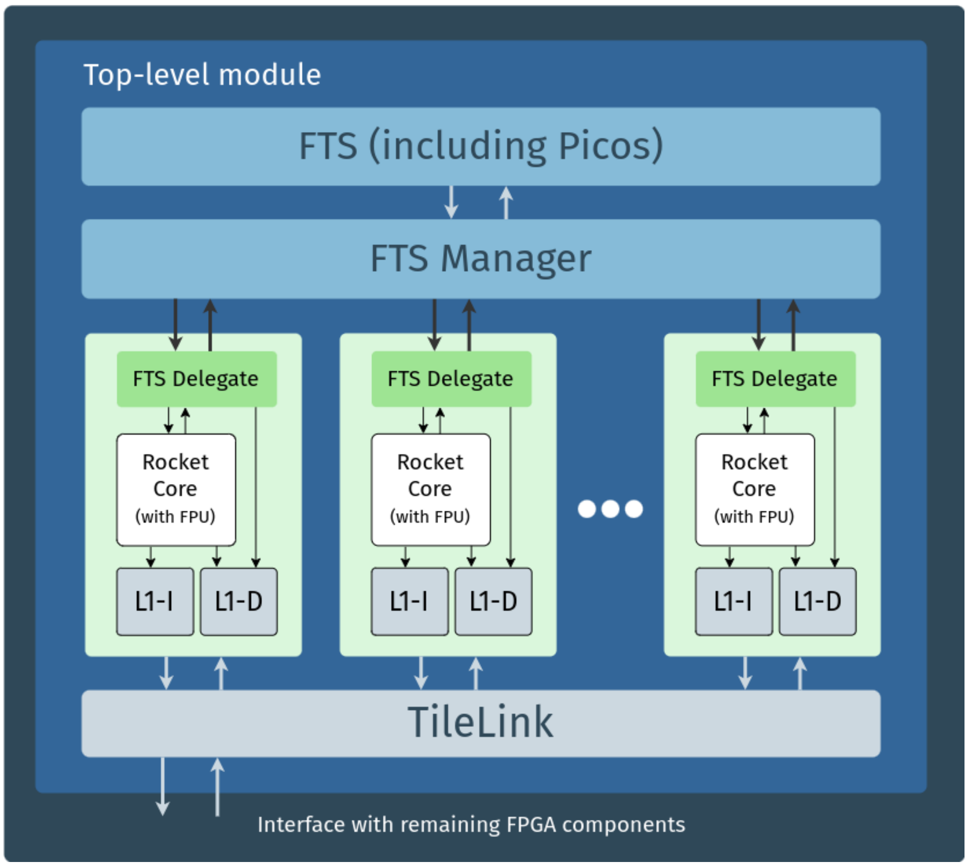


Figure 3.1 Overview of the system architecture

FTS Delegate instances expose Task Scheduling capabilities to individual cores by implementing custom instructions. These instances interact with FTS and Picos through FTS Manager, which arbitrates the distribution of ready-to-run tasks to cores, ensures transaction atomicity, buffers Picos – FTS - 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, FTS Manager, and FTS Delegates can be found throughout the rest of this Section.

3.1 Rocket Chip

We take advantage of Rocket Chip to generate a 30-core RISC-V 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 3.7. Our FPGA prototype includes Rocket Chip instances with relatively large private L1 caches (128 KB for data, 64 KB 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.

4 Custom instruction design

The proposed design relies on the RoCC interface found in Rocket Chip processors. Such interface specifies:

1. A regular parameter format and reserve opcode space for custom instructions.
2. a standard Chisel design pattern for enabling any number of these instructions.
3. a standard interface for connecting independent modules (possibly complex accelerators) implementing the behavior of said instructions.

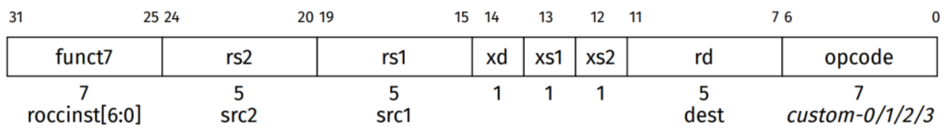


Figure 4.1 RoCC instruction format

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 Figure 4.1. 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. 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 FTS Delegate accelerator, which implements the task-related instructions described at the end of this subsection.

Once the instructions are implemented, user-level libraries might call them using `asm` constructs with string-encoded parameters. User-level applications can then use these accelerators by merely interacting with the library functions wrapping these constructs.

In our system, ten custom instructions were implemented. All custom instructions and their functionality description can be found in table 4.1.

Name	Description
Initiate Task	Informs the system about the task identification 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.

Table 4.1 Description of all custom instructions implemented.

5 The FTS Manager module

FTS Manager arbitrates all data communication between FTS and individual cores. It serves as a protocol converter between the interface defined by core-specific FTS 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 FTS Manager are required, not to the cores.

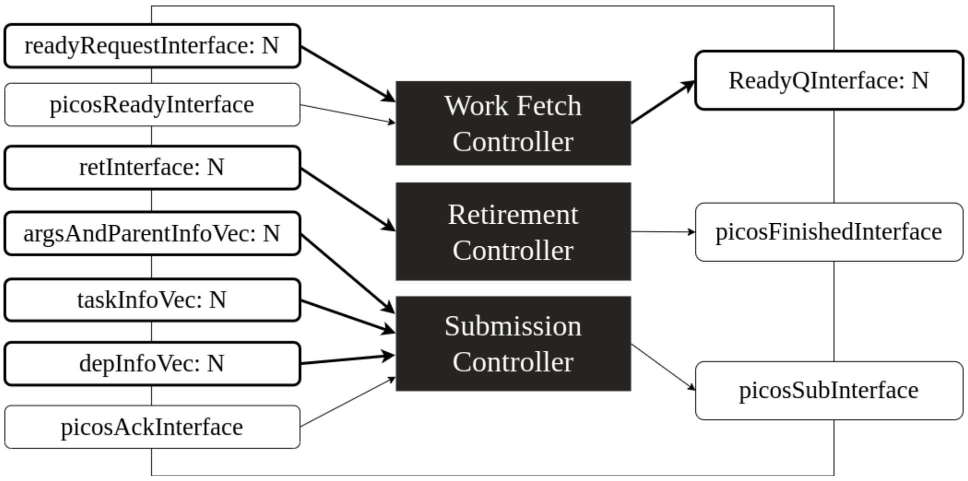


Figure 5.1 The FTS Manager.

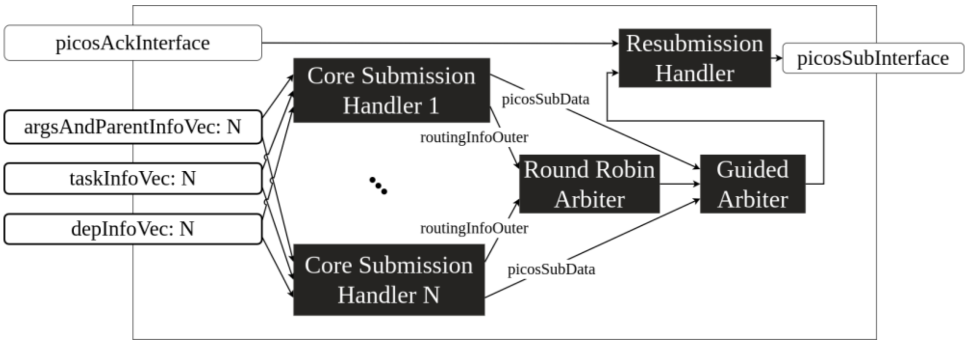


Figure 5.2 Internal block diagram of the Submission Controller.

1. Interface

As shown by Figure 5.1, FTS Manager is connected to FTS and each of the core-specific RoCC accelerators (here called FTS 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 FTS-facing interface includes (5) a ready queue, (6) a retirement queue, and (7) a submission queue.

2. Structural elements

As described by Figure 5.1, FTS 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 Figure 5.2 — is the module that handles processing of submission packets on behalf of FTS 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 to happen atomically; (2) implement protocol crossing logic to ensure that communication between the various cores and Picos comply with Picos interface.

FTS 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 transmits 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.

6 The Phentos runtime

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, signalling of task completion, etc). We shall detail their nature and trade-offs in the following two sub-subsections.

6.1 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.

6.2 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 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). Whenever this is not 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.

6.3 Other design choices to minimize software overheads

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-FTS-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+FTS 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 FTS 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.

6.4 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 Section 4.

Although the system allows for buffering of ready tasks by the cores, both our 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 pass-through channels.

7 Experimental setup

This section describes the characteristics of the system and the benchmarks used to evaluate the FTS IP proposal.

7.1 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

Table 7.1 Summary of system characteristics.

Each experiment is executed on an FPGA instantiation of the system described by Table 7.1. As discussed in Subsection 3.1, 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, 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.

7.2 Benchmarks

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

- The blackscholes application, from the Financial Analysis domain, solves the Black-Scholes partial differential equation for evaluating how the price of a 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.
- 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.
- 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.
- 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.

8 Results and discussion

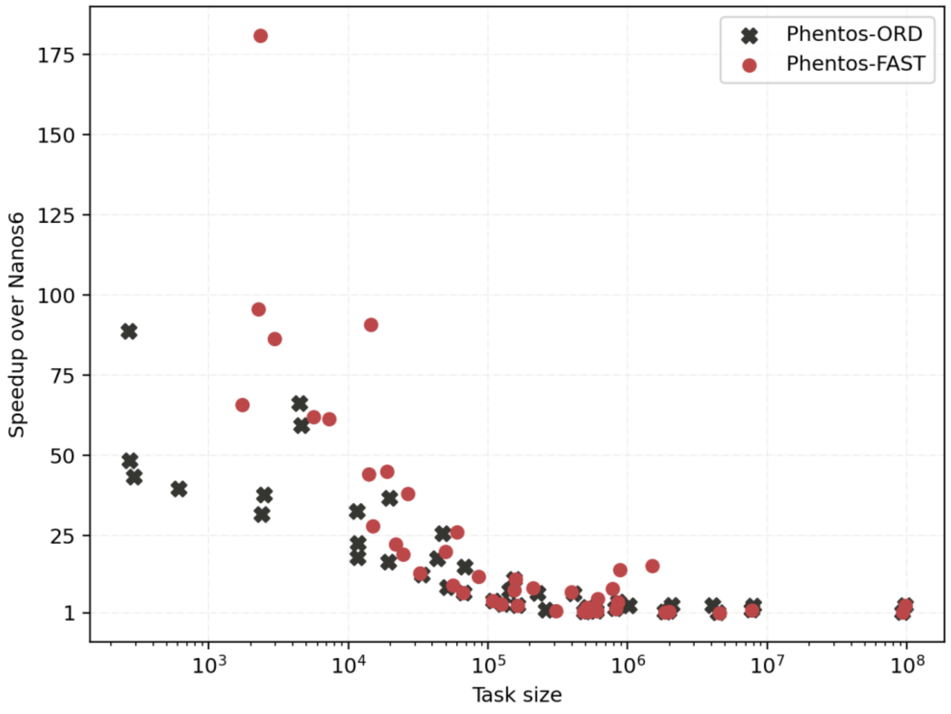
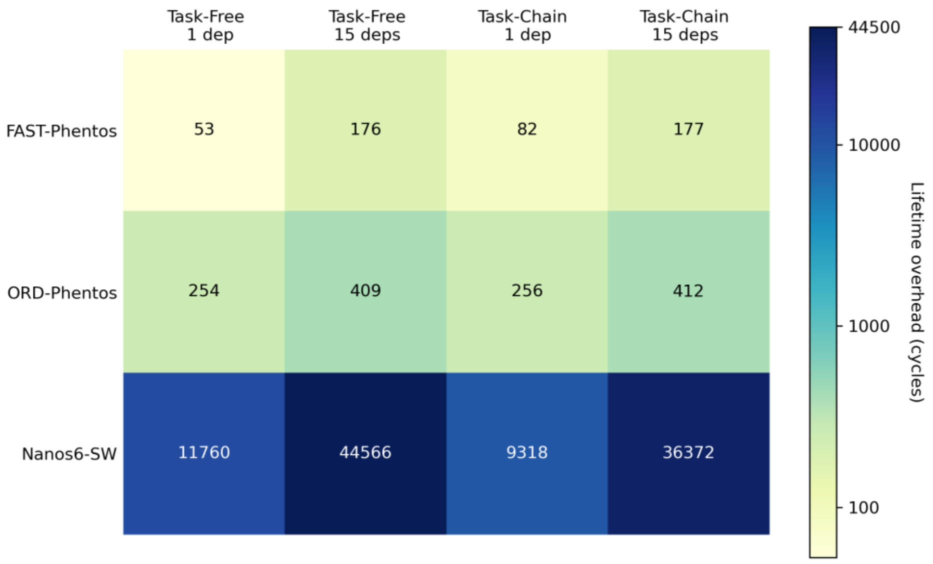


Figure 8.1 Speedups of Phentos executions with respect to equivalent Nanos6 executions of the same workloads, as a function of average application task size in cycles.



	Task-Free 1 dep	Task-Free 15 deps	Task-Chain 1 dep	Task-Chain 15 deps
FAST-Phentos	53	176	82	177
ORD-Phentos	254	409	256	412
Nanos6-SW	11760	44566	9318	36372

44500
10000
1000
100
Lifetime overhead (cycles)

Figure 8.2 Cumulative task-lifetime overhead data for different runtimes and dependence patterns.

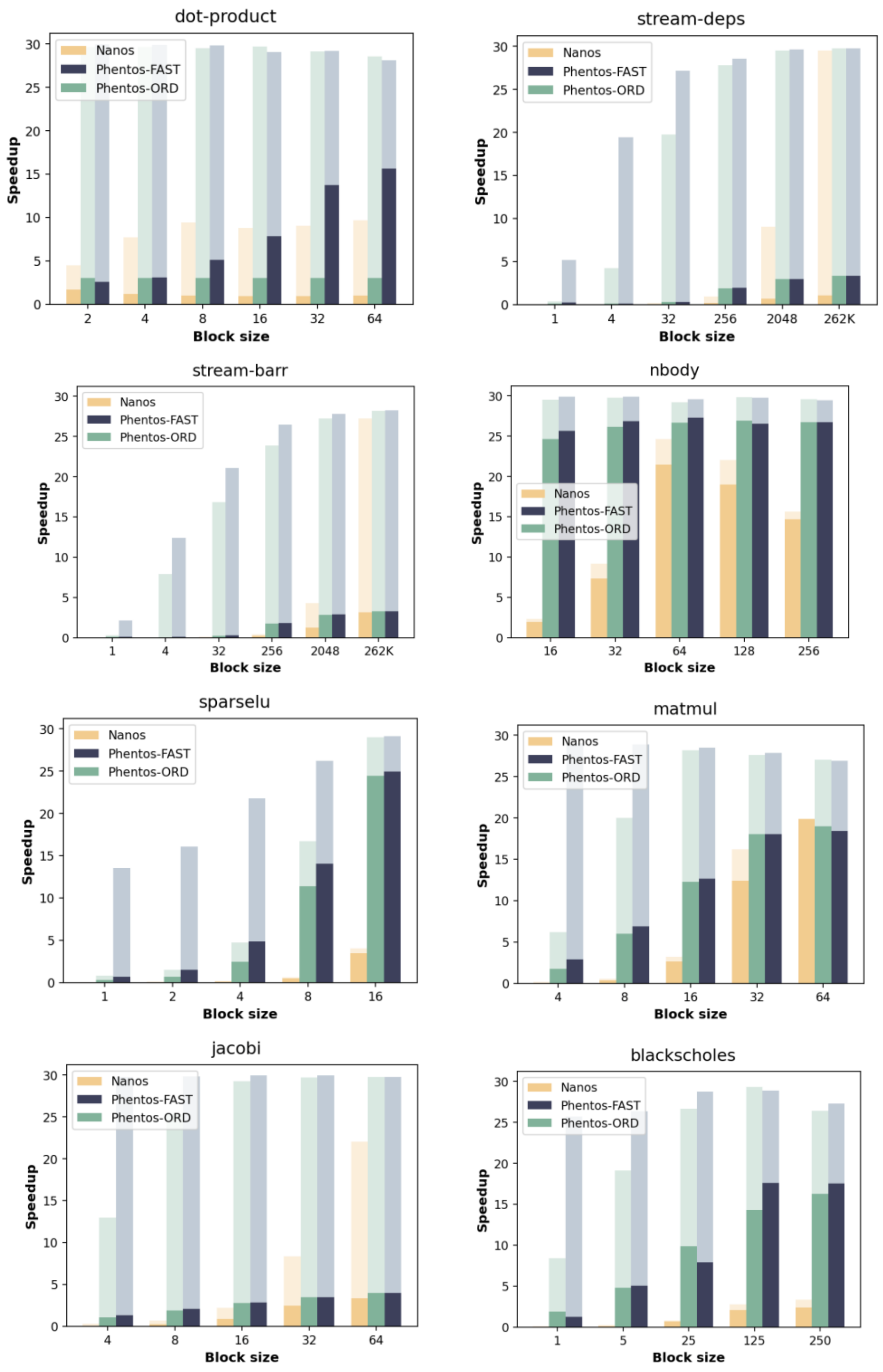


Figure 8.3 Speedup (solid colors) and average core utilization (muted colors) data for various runtimes, benchmarks, and block sizes.

Figure 8.3 summarizes how speedups over serial execution vary with respect to runtime and input selection. We can see that 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.

Figure 8.1 summarizes the Phentos advantage with respect to Nanos, which it clearly suggests being 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 number of tasks per unit of time.

9 Deriving theoretical speedup bounds from MTT

Maximum Task Throughput (MTT) 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. 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:

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

Equation 8.1

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:

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

Having this in mind, for four different workloads, we measured the mean Task Scheduling overhead of NanosRV and Phentos, as shown by Figure 8.2.

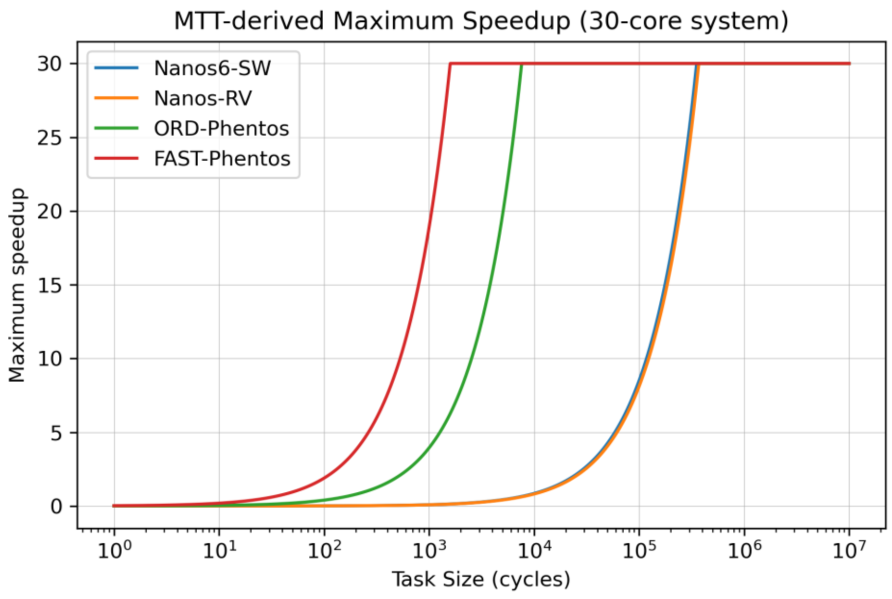


Figure 8.4 Theoretical MTT-derived speedup bounds for several Task Scheduling platforms with 30 cores.

Figure 8.2 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 overhead-measuring benchmarks: Task Free, which generates independent tasks with any number of 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 pointer parameters similarly ranging from 0 to 15. Table 8.1 tells a similar story for tasks with no dependencies.

Based on the figures for the Task-Free (1 dep) case and on Equation 8.1, 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 8.4 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.

Runtime	Dependence-free tasks lifetime latency
FAST-Phentos	32
ORD-Phentos	231
Nanos6-SW	4245

Table 8.1 Lifetime overheads experienced by different runtimes while handling dependence-less tasks.

Finally, we overlay MTT upper bounds to performance data collected for each runtime on Figures 8.5 to 8.7, 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. The crosses in figures 8.5 to 8.7 represent the speedup of the corresponding benchmark point but measuring average core utilization (equivalent to muted colors on figure 8.3).

There, we can see that utilization figures are more likely to be close to MTT limits than over-serial speedups. This is mostly because 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 benchmarks considered, 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 Figure 8.3. 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 are 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.

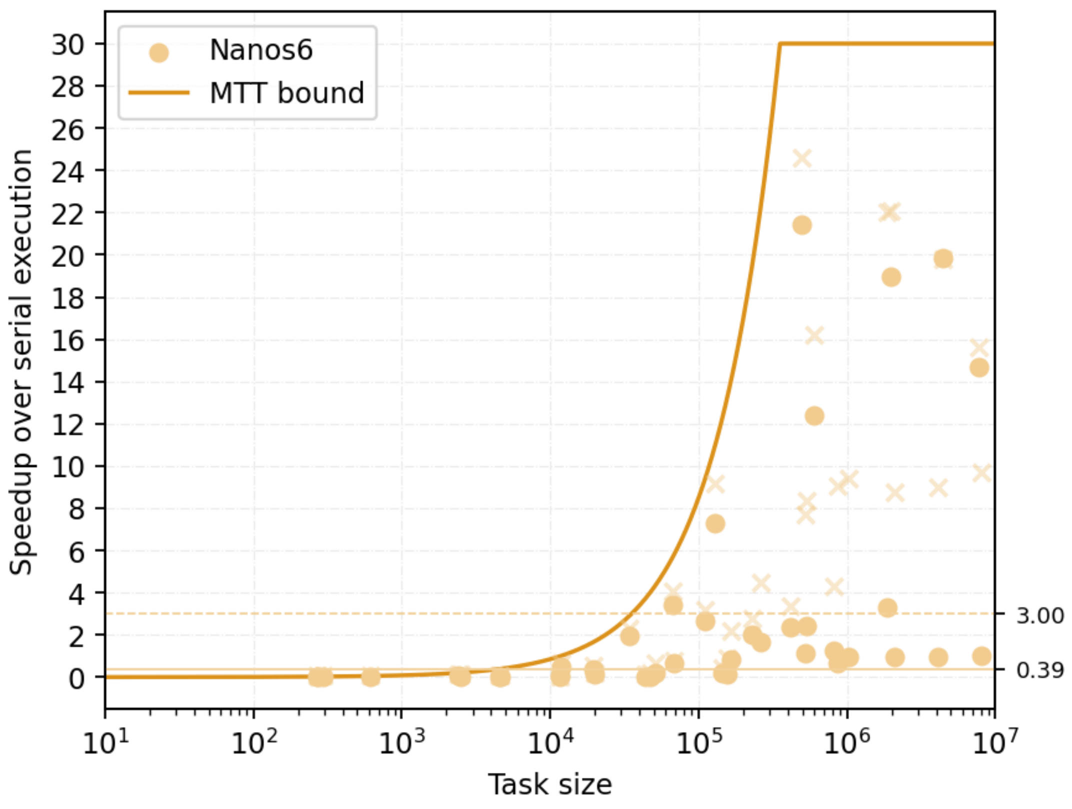


Figure 8.5 Nanos6 application speedups over serial execution as a function of mean task size. The solid and dashed horizontal lines are, respectively, the mean and geomean speedups across all tests.

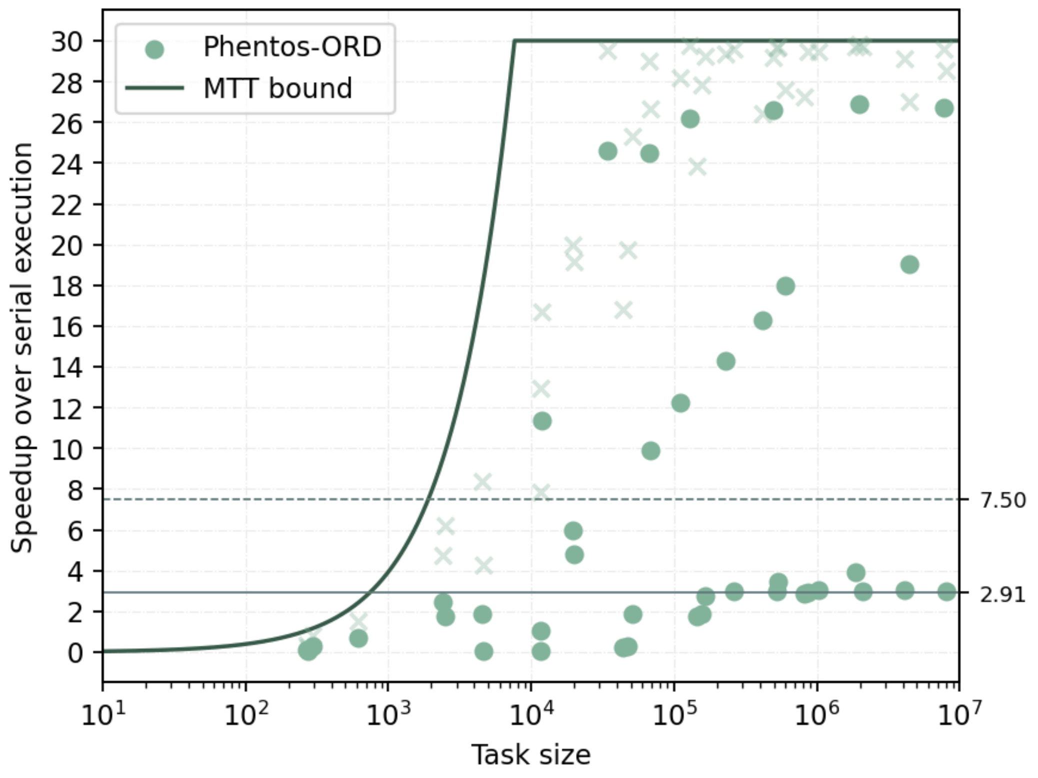


Figure 8.6 Phentos-ORD application speedups over serial execution as a function of mean task size. The solid and dashed horizontal lines are, respectively, the mean and geomean speedups across all tests.

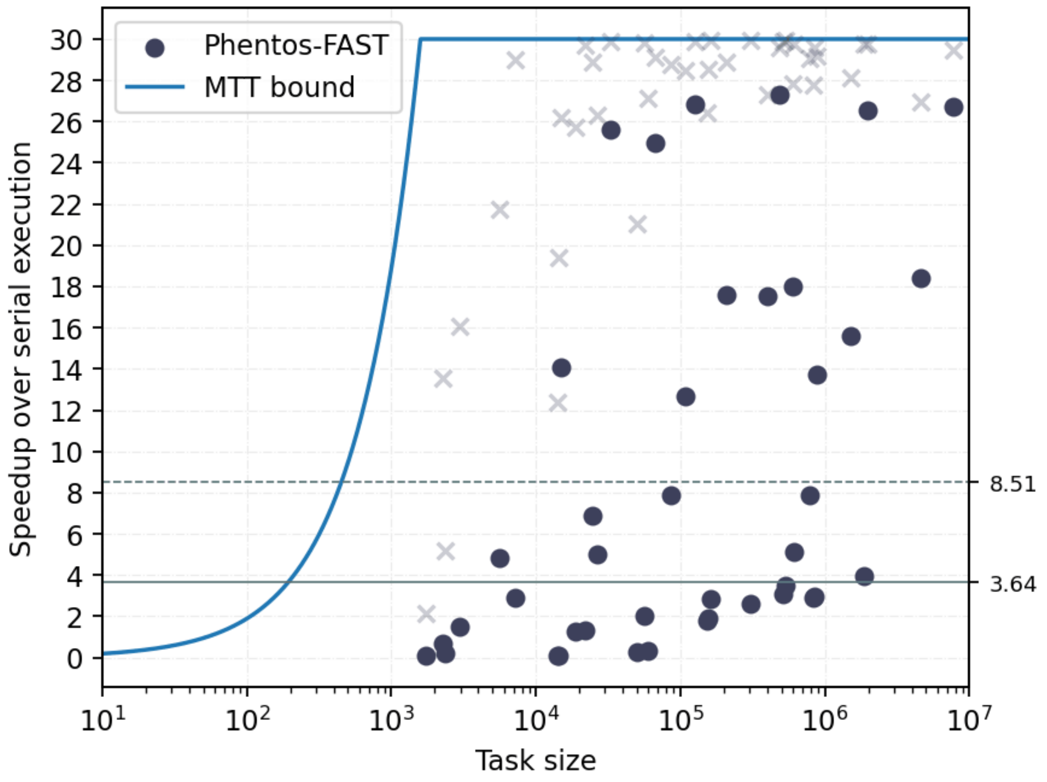


Figure 8.7 Phentos-FAST application speedups over serial execution as a function of mean task size. The solid and dashed horizontal lines are, respectively, the mean and geomean speedups across all tests.

Finally, it is interesting to note that FAST-Phentos data in Figure 8.7 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 causes 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.

9.1 Resource utilization

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%

Table 8.2 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.

Table 8.2 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 whole-design utilization of that resource is due to the Task Scheduling subsystem (comprising Picos, FTS 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.

10 Conclusions and Future Work

As this deliverable shows, the IP for Fast Task Scheduling has been fully developed and behaves as expected. A fully compliant version of the IP has been designed, implemented and tested in the TextaRossa IDV-E platform. The design has been integrated with the OmpSs@FPGA task-based programming model as described in D4.6 Task-based runtime systems and its functionality has been verified.

The Fast Task Scheduler IP has been also integrated into a manycore RISC-V system along with BSC proprietary Dependence Manager IP. The results shown in this deliverable demonstrate that the Fast Task Scheduler IP is fully compliant with a RISC-V hardware architecture and that its design allows great performance improvements over previous state-of-the-art implementations that rely on software to perform the same tasks. The results described here along with a complete research analysis have been published in a Transactions on Computers journal paper under open access format [38].

Our future work that will be carried out in future projects involves further developing the scheduler to improve its intelligence, better optimizing tasks' memory accesses.

11 References

[1] J. Bosch, X. Tan, A. Filgueras, M. Vidal, M. Mateu, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguadé, and J. Labarta, “Application acceleration on fpgas with ompss@fpga,” in International Conference on Field-Programmable Technology, FPT 2018, Naha, Okinawa, Japan, December 10-14, 2018. IEEE, 2018, pp. 70–77.

[2] J. Bosch, M. Vidal, A. Filgueras, C. Álvarez, D. Jiménez-González, X. Martorell, and E. Ayguadé, “Breaking master-slave model between host and fpgas,” in PPoPP ’20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020, R. Gupta and X. Shen, Eds. ACM, 2020, pp. 419–420.

[3] Xubin Tan, Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, Mateo Valero: A Hardware Runtime for Task-Based Programming Models. In IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 9, pp. 1932-1946, 1 Sept. 2019, doi: 10.1109/TPDS.2019.2907493.

[4] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, “Task superscalar: An out-of-order task pipeline,” in Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on. IEEE, 2010, pp. 89–100.

[5] C. Wang, X. Li, J. Zhang, X. Zhou, and X. Nie, “Mp-tomasulo: A dependency-aware automatic parallel execution engine for sequential programs,” ACM Transactions on Architecture and Code Optimization (TACO), vol. 10, no. 2, p. 9, 2013.

[6] S. Kumar, C. J. Hughes, and A. Nguyen, “Carbon: architectural support for fine-grained parallelism on chip multiprocessors,” ACM SIGARCH Computer Architecture News, vol. 35, no. 2, pp. 162–173, 2007.

[7] T. Dallou, N. Engelhardt, A. Elhossini, and B. Juurlink, “Nexus#: A distributed hardware task manager for task-based programming models,” in Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International. IEEE, 2015, pp. 1129–1138.

[8] F. Yazdanpanah, C. Álvarez, D. Jiménez-González, R. M. Badia, and M. Valero, “Picos: A hardware runtime architecture support for ompss,” Future Generation Computer Systems, vol. 53, pp. 130–139, 2015.

[9] Vineyard. (2018) Objectives and rationales of the project. [Online]. Available: http://vineyard-h2020.eu/en/pr

[10] Maxeler, Inc. (2014) The open spatial programming language. [Online]. Available: https://openspl.org

[11] Khronos Group, Inc. (2018) Opencl.Https://www.khronos.org.

[12] Xilinx, Inc. (2020, December) Sdsoc development environment. [Online]. Available: https://www.xilinx.com/sdsoc

[13] Ecoscale Consortium. (2018) Project description. [Online]. Available: http://ecoscale.eu/project-des

[14] A. D. Ioannou, K. Georgopoulos, P. Malakonakis, D. N. Pnevmatikatos, V. D. Papaefstathiou, I. Papaefstathiou, and I.Mavroidis, “Unilogic: A novel architecture for highly parallel reconfigurable systems,” ACM Trans. Reconfigurable Technol. Syst., vol. 13, no. 4, Sep. 2020. [Online]. Available: https://doi.org/10.1145/340911

[15] J. M. Mbongue, D. T. Kwadjo, and C. Bobda, “Automatic generation of application-specific FPGA overlays with rapidwright,” in International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019. IEEE, 2019, pp. 303–306.

[16] C. Lavin and A. Kaviani, “Rapidwright: Enabling custom crafted implementations for fpgas,” in 26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018. IEEE Computer Society, 2018, pp. 133–140.

[17] Xilinx, Inc. (2020, December) Vivado High-Level Synthesis. [Online]. Available: http://www.xilinx.com/hls

[18] Xilinx, Inc. (2020, December) Xilinx Vitis Unified Software Platform. [Online]. Available: https://www.xilinx.com/product

[19] Xilinx, Inc. (2020, December) Sdaccel development environment. [Online]. Available: https://www.xilinx.com/sdaccel

[20] Intel Corp. (2020, December) Intel oneAPI Toolkits. [Online]. Available: https://www.intel.com/oneapi

[21] Intel Corp. (2020, December) Quartus Prime. [Online]. Available: https://www.intel.com/quartus

[22] A. Canis and et al., “LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems,” ACM Transactions on Embedded Computing Systems, vol. 13, no. 2, pp. 24:1–24:27, September 2013.

[23] B. Fort and et al., “Automating the Design of Processor/Accelerator Embedded Systems with LegUp High-Level Synthesis,” in 2014 12th IEEE International Conference on Embedded and Ubiquitous Computing, Aug, pp. 120–129.

[24] Microchip Technology Incorporated. (2020, December) Microchip Acquires High-Level Synthesis Tool Provider LegUp. [Online]. Available: https://www.microchip.com/enus

[25] J. R. Villarreal, A. Park, W. A. Najjar, and R. Halstead, “Designing modular hardware accelerators in c with roccc 2.0.” in FCCM, R. Sass and R. Tessier, Eds. IEEE Computer Society, 2010, pp. 127–134.

[26] W. A. Najjar and J. R. Villarreal, “Fpga code accelerators – the compiler perspective,” in DAC, 2013, p. 141.

[27] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, and D. Jiménez-González, “Openmp extensions for fpga accelerators,” in International Symposium on Systems, Architectures, Modeling, and Simulation, 08 2009, pp. 17 – 24.

[28] L. Sommer, J. Korinth, and A. Koch, “Openmp device offloading to fpga accelerators,” in IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 07 2017, pp. 201–205.

[29] Jaume Bosch, Miquel Vidal, Antonio Filgueras, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé: Task-Based Programming Models for Heterogeneous Recurrent Workloads. In Applied Reconfigurable Computing. Architectures, Tools, and Applications. ARC 2021. Lecture Notes in Computer Science, vol 12700. Springer, Cham. https://doi.org/10.1007/978-3-030-79025-7_8. 2021.

[30] Juan Miguel De Haro Ruiz, Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, Jesús Labarta: OmpSs@FPGA Framework for High Performance FPGA Computing. IEEE Trans. Computers 70(12): 2029-2042 (2021)

[31] Jaume Bosch, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé: Asynchronous runtime with distributed manager for task-based programming models. Parallel Computing, Volume 97, 2020, 102664, ISSN 0167-8191, https://doi.org/10.1016/j.parco.2020.102664.

[32] Jaume Bosch, Miquel Vidal, Antonio Filgueras, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé: PPoPP '20: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. February 2020. Pages 419–420. https://doi.org/10.1145/3332466.3374545

[33} Johannes de Fine Licht, Grzegorz Kwasniewski, Torsten Hoefler: Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis. In FPGA '20: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 2020. Pages 244–254. https://doi.org/10.1145/3373087.3375296

[34] Antonio Filgueras, Daniel Jiménez-González, Carlos Álvarez: Improving resource usage in large FPGA accelerators. 9th BSC Doctoral Symposium Book of Abstracts. 2022.

[35] AMD/Xilinx: LogiCORE IP Block Memory Generator v7.1 Data Sheet (DS512) https://docs.xilinx.com/v/u/en-US/blk_mem_gen_ds512

[36] ARM: AMBA 4 AXI4-Stream Protocol Specification (ihi0051) https://developer.arm.com/documentation/ihi0051/a/

{37} Lucas Morais, Vitor Silva, Alfredo Goldman, Carlos Álvarez, Jaume Bosch, Michael Frank, Guido Araujo: Adding Tightly-Integrated Task Scheduling Acceleration to a RISC-V Multi-core Processor. MICRO 2019: 861-872

[38] L. Morais et al., "Enabling HW-based Task Scheduling in Large Multicore Architectures," in IEEE Transactions on Computers, doi: 10.1109/TC.2023.3323781.

Appendix A. Hardware modules source code

Source code of all hardware modules described in this document, as well as the wrappers that interconnect and instantiate them, are available via BSC's B2Drop platform:

https://b2drop.bsc.es/index.php/s/tbEzqEHegxNXLP6

Also, the source code of the hardware can be found integrated in the OmpSs@FPGA framework in the OmpSs@FPGA public Github page:

https://github.com/bsc-pm-ompss-at-fpga

The most recent implementation of the fast task scheduler is integrated in the OmpSs-2@FPGA release that is the version currently under development (OmpSs@FPGA is no longer updated, only bug fixes are applied to it).

Appendix B. Queues and Commands information

The following section describes the structure of memories used to communicate the Host (usually using libxtasks) with the FTS.

Command in and Command out queues

Each queue has 1024 elements (uint64_t type) and it is divided into 16 subqueues of 64 elements. Each subqueue corresponds to one accelerator, starting from accelerator 0 (positions [0,63]) to accelerator 15 (positions [960,1023]) as shown in Figure B.1.

```
 1023              64 63             0
+---------------------------------------+
| | | | |    ...    | | |    ...    | | |
+---------------------------------------+
                      <-- 1 subqueue -->
<----------- 1024 positions ----------->
```

Figure B.1 Command in and Command out queues.

Each command uses a dynamic number of slots in the queue. The number of slots depends on the command. The odd command codes make the accelerator become busy (no further commands will be sent to the accelerator until it returns the command out response) and the even command codes do not. The information is structured as shown in Figure B.2.

```
 63                                                          0
+-----------------------------------------------------------------+
| Valid |              Command Arguments               | Code   |
+-----------------------------------------------------------------+
|                       Command payload                           |
|                            ...                                  |
+-----------------------------------------------------------------+
<--------------------- 64 bits - 8 bytes ---------------------->

 - [7  :0  ] Command code
   *  0x01 - Execute task cmd
   *  0x03 - Finished task cmd
   *  0x05 - Execute period task cmd
 - [55 :8  ] Command arguments
 - [63 :56 ] Valid Entry
   *  0x00 - Invalid
   *  0x80 - Valid
 - [   :   ] Command payload
```

Figure B.2 Commands format.

As it can be seen the commands use the first position to indicate the command and the next positions as a payload (actual command information).

Commands format

We have defined three initial commands in the FTS, a Execute task command, a Finished task command notification and a Execute periodic task command. The Execute task and Execute periodic task commands follow the structure shown in Figure B.3.

```
 63                                                            0
+-----------------------------------------------------------------+
| Valid |       | DesID | CompF |                | #Args | Code   |
+-----------------------------------------------------------------+
| 0x00  |               Task Identifier                           |
+-----------------------------------------------------------------+
|                    Parent Task Identifier                       |
+-----------------------------------------------------------------+
|              Period           |        Num. repetitions         |
+-----------------------------------------------------------------+ ^
|            ArgumentID         |                        | Flags  | |
+-----------------------------------------------------------------+ | 1 arg
|                             Argument                            | |
+-----------------------------------------------------------------+ v
|                          Other arguments                        |
|                                ...                              |
+-----------------------------------------------------------------+
<--------------------- 64 bits - 8 bytes ---------------------->

 - [7  :0  ] Command code
    *  0x01 - Execute task cmd
    *  0x05 - Execute periodic task cmd
 - [15 :8  ] Number of arguments
 - [31 :16 ]
 - [39 :32 ] Compute flag
    *  0x00 - Compute disabled
    *  0x01 - Compute enabled
 - [47 :40 ] Destination ID where the accelerator will send the 'complete'
signal
    *  0x1F - Processing System (PS)
 - [55 :48 ]
 - [63 :56 ] Valid Entry
   *  0x00 - Invalid
   *  0x80 - Valid
 - [119:64 ] Task identifier
 - [127:120] 0x00 constant. This field is used to identify task commands
created externally
 - [191:128] Parent Task identifier. This field is ignored by the FTS and the
accelerators, it is maintained to match the format of the internal command
queue and the format expected by the accelerators.
 - [223:192] Number of times that task body will be executed (Execute periodic
task cmd only)
 - [255:224] Time (us) between task body launches (Execute periodic task cmd
only)

Each argument is:
 - [7  :0  ] Flags
    *  0x00 - BRAM
    *  0x01 - Private
    *  0x02 - Global
    *  0x10 - Enable input copy to wrapper BRAM
    *  0x20 - Enable output copy from wrapper BRAM
    * bit7 is internally used by cmd In module to store whether the input
copy has been optimized or not.
 - [31 :8  ]
 - [63 :32 ] Argument ID
 - [127:64 ] Argument value
```

Figure B.3 Execute task and Execute periodic task commands format.

Finally, Figure B.4 shows the Finished task command format. As it can be seen this command is simpler as it only sends the task identifier information. This information is used by the task creator (runtime running in the host) to keep track of possible dependencies and could also be used by the FTS to identify the accelerator that has finished.

```
 63                                                             0
+-----------------------------------------------------------------+
| Valid |                                                | Code   |
+-----------------------------------------------------------------+
|                         Task Identifier                         |
+-----------------------------------------------------------------+
<---------------------- 64 bits - 8 bytes ----------------------->

 - [7  :0  ] Command code (value fixed to `0x03`)
 - [55 :8  ]
 - [63 :56 ] Valid Entry
   *  0x00 - Invalid
   *  0x80 - Valid
 - [127:64 ] Task identifier sent to the accelerator in the execute task
command
```

Figure B.4 Finished task command format.

Appendix C. IP core design

This section describes the basic IP design, interface with accelerators and host system and IP operating modes and parameters. Command interfaces with the software part of the system are described in finer-grain detail in D2.1 Consolidated specs of accelerators IPs and copied in Appendix B of this deliverable for convenience.

Basic IP workflow design

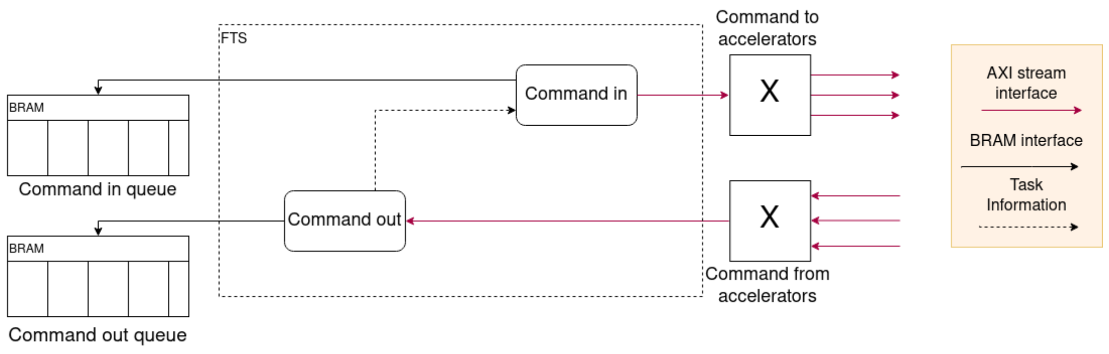


Figure C.1 IP for fast task scheduling diagram (FTS).

Figure C.1 shows a diagram of the IP for fast task scheduling (from now on FTS or Fast Task Scheduler). The main objective of the FTS is to take care of scheduling tasks into individual accelerators. To achieve this objective, the FTS IP is composed of two command queues, one for input coming from the CPU/exterior of the FPGA ("Command in queue") and another going to the CPU/exterior of the FPGA ("Command out queue"), two control modules ("Command in" and "Command out") and two interconnection multiplexers/demultiplexers.

The workflow in the FTS follows. First of all, tasks are sent from the host CPU to the Fast Task Scheduler by using commands that are temporarily stored in the "Command in queue". These commands are processed in order by the "Command in" module and, depending on the accelerator's availability, are sent to the appropriate one. Commands are sent through the "Command to accelerators" demultiplexer through an AXI stream interface, and only when accelerators are available (ready) in order to avoid interface contention and starvation.

Once the task has been processed by the corresponding accelerator, the accelerator informs the FTS through an output AXI stream interface that is multiplexed ("Command from accelerators") to reach the "Command out" module with a "Finished Task" command. "Finished Task" command is expected to be processed in very few cycles (tens of cycles at most). Therefore, although some contention can be expected when several accelerators finish at the same time submitting this command, no significant performance drop is expected in this case. The "Command out" module is in charge of processing the "Finished Task" packet by forwarding it with the adequate format to the "Command out queue" and to notify the "Command in" module about the new ready state of the accelerator for the FTS to forward a new task to it.

Interface port descriptions

IP core ports are shown in Figure C.2. Other than the clk and reset signals, there are two classes of ports: AXI4 stream (cmdout_in, cmdin_out), and BRAM interface (cmdin_queue, cmdout_queue).

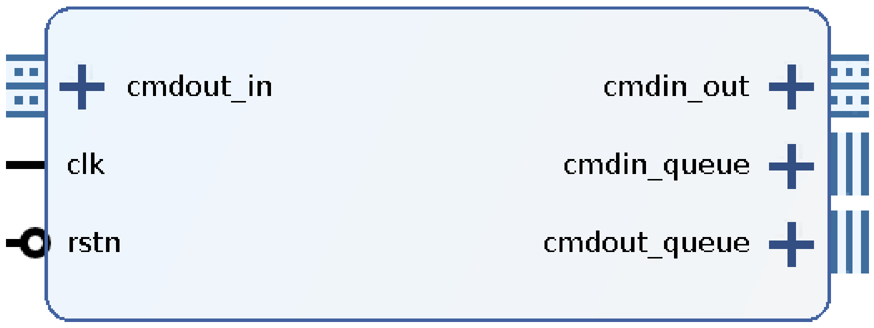


Figure C.2: Fast Task Scheduler I/O ports

Detailed description of each port as well as individual signal descriptions are provided in the following sections.

Cmdout_in Port description

Cmdout_in is an AXI stream slave port that receives commands from accelerators, such as finished tasks. Detailed description of the command format is specified in deliverable D2.1 and copied in Appendix B of this document for convenience.

Cmdout_in Signal description

Signals conforming this port are a subset of the standard AXI Stream [9] interface signals for a slave interface, which are described in table C.1.

Name	I/O	Width	Description
tid	I	$\lceil \log_2 \text{MAX\_ACCS} \rceil$	Identifies the origin accelerator of the command
tdata	I	64	Stream data payload
tvalid	I	1	Data set by the master interface is valid
tready	O	1	Slave interface is ready to receive data

Table C.1: cmdout_in signal description

Notes:

- A data transfer takes place when tvalid and tready signals are asserted to 1 at the same time.
- MAX_ACCS specifies the maximum number of accelerators supported in current design.

Cmdin_out Port description

Cmdin_out is an AXI stream master interface. It sends out input commands, to accelerators in the design. Accelerator input commands are described in deliverable D2.1 and copied in Appendix B of this document for convenience.

Cmdin_out Signal description

Signals in this port are a subset of the AXI Stream [9] interface signals for a slave interface, which are described in table C.2:

Name	I/O	Width	Description
tdest	O	[$\log_2$ MAX_ACCS]	Destination accelerator for the current transaction
tdata	O	64	Stream payload data
tlast	O	1	Signals the last element of the current transaction
tvalid	O	1	Current payload data is valid
tready	I	1	Slave interface is ready to receive data.

Table C.2: cmdin_out signal description

Notes:

- A data transfer takes place when tvalid and tready signals are asserted to 1 at the same time.
- MAX_ACCS specifies the maximum number of accelerators supported in current design, see section 2.3 for more details on MAX_ACCS.
- `tlast` is an should be asserted on last data beat of a command. This is needed by the stream interconnection in order to correctly route stream traffic.

Cmdin_queue Port description

This port is a BRAM interface that connects the FTS IP core to the command in circular queue. Data layout for this queue is defined in document D2.1

Cmdin_queue Signal description

Signals belonging to this bus implement a Xilinx BRAM interface [8] and are described in table C.3

Name	I/O	Width	Description
en	O	1	Enables read, write and reset operations
dout	I	64	Data ouput from read operations
din	O	64	Data input for write operations
we	O	8	Write enable for individual bytes of din
addr	O	32	Memory addres to read or write
clk	O	1	Bus clock, all operations are synchronous to this clock
rst	O	1	BRAM active-high reset

Table C.3: cmdin_queue signal descriptions (I/O directions under FTS IP core point of view)

Notes:

- A data read is performed when `en` signal is asserted
- A data write is performed when `en` and `we[n]` are asserted, in which case, only n-th byte will be written to memory.
- Multiple bytes can be written at the same time by setting multiple `we` bits to 1

CmdOut_queue Port description

This port connects the FTS IP core to the `command_out` circular queue BRAM memory. Data layout for the command out queue is defined in document D2.1

CmdOut_queue Signal description

Signals belonging to this bus implement a xilinx bram interface [8] and are described in table C.4

Name	I/O	Width	Description
en	O	1	Enables read, write and reset operations
dout	I	64	Data ouput from read operations
din	O	64	Data input for write operations
we	O	8	Write enable for individual bytes of din
addr	O	32	Memory address to read or write
clk	O	1	Bus clock, all operations are synchronous to this clock
rst	O	1	BRAM active-high reset

Table C.4: cmdout_queue signal descriptions (I/O directions under FTS IP core point of view)

Notes:

- A data read is performed when `en` signal is asserted
- A data write is performed when `en` and `we[n]` are asserted, in which case, only n-th byte will be written to memory.
- Multiple bytes can be written at the same time by setting multiple we bits to 1

Clk Signal description

Clk is the input clock signal for the FTS IP core. All stream interfaces (cmdin_out, cmdOut_in) are synchronous to this clock signal and therefore matching master or slave interfaces must be in the same clock domain.

Synthesis has been verified up to 300MHz in Xilinx UltraScale+ devices, but higher frequencies may be achievable on higher speed grade devices under certain conditions.

Rstn Signal description

Rstn is the active low reset signal for the FTS core. It resets the core to a known state after being deasserted for more than one cycle. During normal operation this signal should be asserted to 1 as it is an active low reset. This reset signal is synchronous to the clk signal.

Operating Modes & Parameters

In order to customize the FTS for a particular design, some features can be parametrized. This allows internal data structures to be tailored for the current design, allowing the system to save resources as only needed sources will be allocated.

Table C.5 shows a list and brief description of configuration parameters

Name	Allowable values	Default value	Type	Description
MAX_ACCS	2-8192	16	Integer	Maximum number of accelerators supported

MAX_ACC_TYPES	2-8192	16	Integer	Maximum number of different accelerator types supported
CMDIN_QUEUE_LEN	4-8192	64	Integer	Length of each accelerator command in queue
CMDOUT_QUEUE_LEN	4-8192	64	Integer	Length of each accelerator command out queue
MAX_ARGS_PER_TASK	0-8192	15	Integer	Maximum number of arguments in a task

Table C.5: FTS parameter list

Notes:

- Even though FTS allows a maximum of 8192 accelerators and accelerator types, there are practical limits on the number of accelerators that may fit in a particular device, that usually are well below this number
- CMDIN_QUEUE_LEN and CMDOUT_QUEUE_LEN define the length of each accelerator sub-queue; total queue size is defined by MAX_ACCS*CMDIN_QUEUE_LEN or MAX_ACCS*CMDIN_QUEUE_LEN.
- BRAM attached to cmdin_queue or cmdOut_queue must be large enough to fit to store all accelerators sub queues.