Performance Analysis of a Hardware Accelerator of Dependence Management for Task-based Dataflow Programming models

Xubin Tan*, Jaume Bosch†, Daniel Jiménez-González‡, Carlos Álvarez-Martínez§, Eduard Ayguadé¶ and Mateo Valero‖
Barcelona Supercomputing Center, Universitat Politécnica de Catalunya, Barcelona, Spain
Email: {*xubin.tan, †jbosch, ¶eduard, ‖mateo.valero}@bsc.es, {‡djimenez, §calvarez}@ac.upc.edu

***Abstract*—Along with the popularity of multicore and many-core, task-based dataflow programming models obtain great attention for being able to extract high parallelism from applications without exposing the complexity to programmers. One of these pioneers is the OpenMP Superscalar (OmpSs). By implementing dynamic task dependence analysis, dataflow scheduling and out-of-order execution in runtime, OmpSs achieves high performance using coarse and medium granularity tasks. In theory, for the same application, the more parallel tasks can be exposed, the higher possible speedup can be achieved. Yet this factor is limited by task granularity, up to a point where the runtime overhead outweighs the performance increase and slows down the application.**

To overcome this handicap, Picos was proposed to support task-based dataflow programming models like OmpSs as a fast hardware accelerator for fine-grained task and dependence management, and a simulator was developed to perform design space exploration. This paper presents the very first functional hardware prototype inspired by Picos. An embedded system based on a Zynq 7000 All-Programmable SoC is developed to study its capabilities and possible bottlenecks. Initial scalability and hardware consumption studies of different Picos designs are performed to find the one with the highest performance and lowest hardware cost. A further thorough performance study is employed on both the prototype with the most balanced configuration and the OmpSs software-only alternative. Results show that our OmpSs runtime hardware support significantly outperforms the software-only implementation currently available in the runtime system for fine-grained tasks.

I. Introduction

With increasing usage of chip multiprocessors (CMPs), we face a variety of challenges leading to the path of parallel computing. The basic premise is simple: the more parallel computations/tasks we can expose in our application, the more workers we can employ and finish faster. The reality though, is much more complicated. Detecting parallel regions, distributing tasks evenly and synchronizing between different tasks are costly both in software and in hardware, and burdensome for the programmer. As the scale of CMPs keeps increasing, the reality is likely to outweigh the performance benefit. One direct way to tackle those challenges is using task-based programming models. For example, Google's MapReduce [1], Intel's TBB [2], OpenMP 4.0 [3], and StarSs [4], are programming models that allow extracting parallelism from applications with small effort for programmers. In this paper, we use the OmpSs programming model [5] which is developed based on both OpenMP and StarSs. The OmpSs programming model employs dataflow principles by abstracting the application as a collection of tasks. In the source code you can specify a task with its data dependences (input, output, inout) [6]. The runtime automates the inter-task synchronization and parallelism by dynamic task dependence analysis, dataflow scheduling and out-of-order execution. OmpSs is able to expose high parallelism from applications of varied domains using coarse and medium granularity tasks, with both regular and irregular task dependence patterns.

As the number of processors of multicore and manycore continues to grow, an increasing amount of tasks are required to make full usage of the available hardware resources. However, for fine-grained tasks, the overhead of a software-only implementation including task creation, dependence management, task scheduling, etc., is simply too high to allow it to maintain a scalable performance [7]. Figure 1 shows the

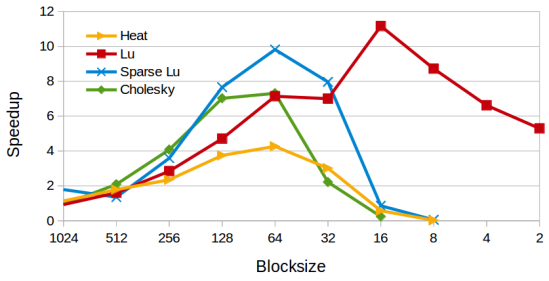


Fig. 1. Speedup vs Task Granularity

speedup (y-axis) obtained for four OmpSs parallel applications compared to a sequential execution by using 12 cores and an OmpSs software-only implementation. For all the applications - Gauss-Seidel Heat, Lu, SparseLu and Cholesky [8], their problem sizes are constant and their blocksizes are decreasing. The number of tasks (available parallelism) grows exponentially as the task granularity (blocksize, x-axis) decreases. The overall speedup first increases due to new potential parallelism available, then decreases after certain point when the overhead surpasses these gains.

To overcome this deficiency and improve the performance, a straightforward and effective way is to reduce the software-only runtime overhead. Note that while reducing the task

creation overhead is important, accelerating the task and dependence management is far more crucial [9]. Task Superscalar [10] was proposed to accelerate task and dependence management using hardware. Its first VHDL prototype simulation analysis using ModelSim demonstrated high potentials [11] by employing inter-task dependence analysis, dependence renaming and out-of-order execution. However, its straightforward hardware implementation presented unresolved deadlocks due to queue saturation and memory capacity. A new design called Picos [7] was proposed and simulated with a C simulator to improve Task Superscalar by resolving these deadlocks and adding support for nested tasks. In this paper, we present a hardware accelerator for task and dependence management of fine-grained tasks for task-based programming models. The study builds on the prior Picos model, and deadlocks are corrected using a new operational workflow and new DM designs (described in Section III). And it is the first successfully prototype realized in hardware and integrated within an embedded ARM processor in FPGA.

- Design Exploration of different configurations of Picos with the objective of identifying the best design to reduce the latency of task and dependence management, meanwhile increasing the task execution throughput.
- Proof-of-concept of functional hardware implementations for all Picos configurations analyzed. They are implemented inside a Zynq 7000 All-Programmable SoC.
- Evaluation of all the hardware designs presented. This evaluation includes scalability for synthetic and real applications, resource consumption, and comparison to the current software runtime library of OmpSs.

The paper is organized as follows: Section II briefly introduces the OmpSs programming model and reviews related work. Section III presents the hardware architecture of Picos and its operational flow in detail. Section IV describes the experimental setup, embedded system and benchmarks in use. In Section V, detailed hardware consumption and performance of different Picos designs are analyzed, latency and throughput of the prototype are studied, and scalability between the Picos prototype and the OmpSs software-only implementation is evaluated. Finally, Section VI and VII present the main lessons and conclusions of this work.

II. Background

A. OmpSs Programming Model

The OmpSs programming model [5], [6] provides a simple and powerful way of annotating sequential programs with simple directives to exploit heterogeneity and task parallelism. For example, in C language: `#pragma omp task [input (...)] [ output (...)] [ inout (...)]` is used to specify a task with the direction of its dependences (scalars or arrays). The Mercurium source-to-source compiler transforms programs with those directives into a parallel application; and the Nanos++ Runtime System (RTS) provides services to manage task creation, dependence analysis and task scheduling, etc. Implicit synchronization between tasks is automatically managed through dependence analysis, and explicit synchronization is managed by using `#pragma omp taskwait` and `#pragma omp taskwait on (...)`. `taskwait` makes the thread wait until all its child tasks finish before it can create new tasks; and `taskwait on` ensures that the thread waits until the associated dependence is realised before new tasks can be created.

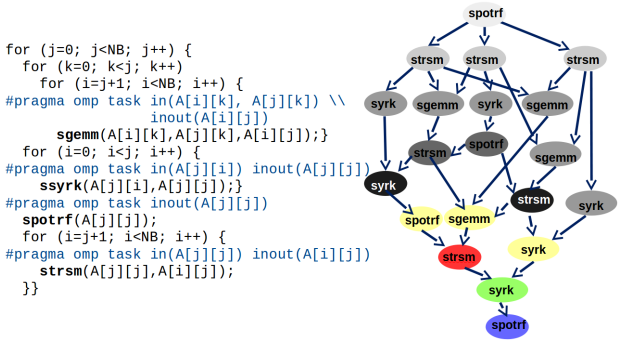


Fig. 2. Cholesky Factorization

Figure 2 shows an example of OmpSs Cholesky Factorization source code and a task dependence graph managed by the Nanos++ RTS. When a function annotated with pragma is called, a new Task Work Descriptor including task identification, the number, memory addresses and directions of dependences is created; dependence analysis is then performed on created tasks, once all the dependences of one task are ready, it can be scheduled to an idle worker. Assuming all tasks are the same size, one possible parallel execution is shown for a 6 cores machine (tasks with the same color are run in parallel).

B. Related Work

A large amount of research work has focused on hardware support for task and dependence management.

Intel CARBON [12] introduces a hierarchy hardware queue architecture and employs task stealing to speedup task dispatching and retrieving. It uses a centralized global task unit (GTU) to enqueue and dequeue tasks per thread, and a smaller local task unit (LTU) for buffering tasks per core. Asynchronous Direct Messages (ADM) [13] presents a combined SW/HW approach which expands on the CARBON architecture by introducing scheduling extensions in software to cater different kinds of applications. Task Scheduling Unit [14] is another hardware queue architecture that accelerates task scheduling. These works focus more on task management approaching task scheduling, other aspects as inter-task dependences and synchronizations are performed by the programmer. In contrast, the Picos Hardware focuses on both dynamically detecting inter-task dependences and scheduling tasks automatically, transparently to the programmer.

A video-oriented task scheduler [15], is proposed as a hardware support for fast task creation, synchronization and scheduling. Similar mechanisms have also been applied by the programmable Task Management Unit (TMU) [16]. These

works achieve high performance for specific inter-task dependence patterns in video processing domains. Another interesting design is the Multilevel Computing Architecture (MLCA) [17], which introduces a novel architecture focusing on multimedia multicore systems for coarse-grained task parallelism. The MLCA augments a traditional muticore architecture (served as low level) with a CP (Control Processor, as high level). The CP employs task queue, register renaming, out-of-order execution to dynamically detect the inter-task dependence and schedule tasks when they are ready. All these research works focus on developing hardware for applications in a specific domain, while the Picos Hardware aims at providing a general support for applications of different domains.

Nexus# [18] is proposed to accelerate task and dependence management for the OmpSs programming model in hardware. It employs similar mechanisms as Task Superscalar [11] while using less hardware. Hardware cost for the Picos prototype in this paper is around 38% of Nexus# 1TG and 18% of Nexus# 2TGs. Application H264dec video decoder with four decreasing task sizes (8x8, 4x4, 2x2, 1x1) is used for performance comparison. The speedup of the Picos prototype in Section V-D against Nexus# 1TG is 2.4x, 1.1x, 1x, 1.2x, and against Nexus# 2TGs is 1.2x, 0.8x, 1x, 1.2x with 24 cores. To summarize, Picos prototype yields competitive performance with Nexus# with much lower hardware cost.

Swarm[19] is a novel architecture to exploit ordered irregular parallelism in task-based parallel applications. It addresses Transactional Memory (TM) database applications based on Thread-level Speculation (TLS), and gains high performance by speculative execution supported by the co-design of the execution model and microarchitecture. The execution model uses timestamps specified on tasks by the programmer to reduce false data dependences and the microarchitecture supports large speculation window, selective aborts and ordered commits. Picos Hardware manages data dependences in a dataflow manner, without the need for TM and speculation.

III. Hardware Description

Picos Hardware aims at accelerating task and dependence management of fine-grained tasks for task-based data-flow programming models. From the software aspect, it can be seen as a co-processor that (1) receives task dependence information (task id and its dependences) at task creation time, and (2) sends ready-to-execute task information to the worker threads. We describe the Picos Hardware in a top-bottom fashion way, first present its organization, second describe its operational flow, and finally focus on critical path designs in detail.

A. Hardware Organization

We present the organization of the Picos Hardware in Figure 3a and the current implementation in Figure 3b. The Picos Hardware is composed of one Gateway (GW), one Task Scheduler (TS), one Arbiter (ARB), and N instances of the Task Reservation Station (TRS) and of the Dependence Chain Tracker (DCT). This architecture is scalable by simply increasing the number of TRSs and DCTs. A design with four instances is able to manage up to 256 cores, and a baseline configuration with only one TRS and DCT is able to manage up to 8 cores without significant performance loss based on simulation results [7]. Functional descriptions of each component are as follows:

GW: the first interface between Picos and the processing cores. It fetches new tasks from memory and finished tasks from workers, and dispatches them to TRSs and DCTs.

TRS: the major task management unit. It stores in-flight tasks, tracks the readiness of new tasks and manages the deletion of finished tasks. There is a Task Memory (TM) per TRS to store task identification (`Task.ID`), the number of dependences per task (`#Num.Dep.`), and consumer sections to store inter-task dependences detected and notified by DCT. In Figure 3b, TM0 is used to store the `Task.ID`, `#Num.Dep.` and the number of ready dependences (`#Ready Dep.`), the last two are used to count the number of ready notifications from DCT for TRS to mark the task ready. TMXs are used to store consumer sections information notified by DCT.

DCT: the major dependence management unit. It manages task dependences through one Dependence Memory and one Version Memory (DM and VM respectively) per DCT. Each DM stores the memory address of one dependence (In Figure 3b, `Tag`) and performs address matching against the addresses of those arrived earlier, to track data dependences. Since each dependence is saved only once in DM, VM is used to save and control all its live versions (`Consumer, TRS_slot` for Producer-Consumer dependence, and `Producer, TRS_Slot` for Producer-Producer dependence). DM and VM also stores pointers to each other and counters for dependences that have the same address or are consumers in addition respectively.

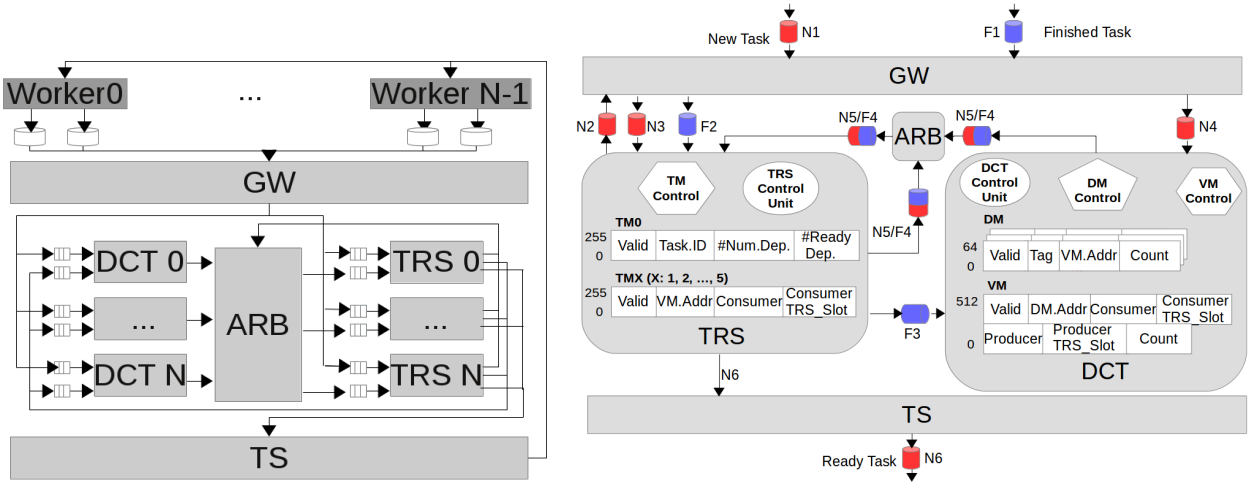
ARB: manages communications between TRSs and DCTs.

TS: the second interface between Picos and the processing cores. It stores ready tasks and schedules them to idle workers. In the current implementation, FIFO queue is the default task scheduling method.

In Figure 3b, each component has its own control unit, which only relies on the status (empty or full) and packets of those FIFOs to ensure asynchronous communications with other modules. TM0 has 256 entries, each entry saves information for one task, these enable it to manage up to 256 in-flight tasks. TMXs (X: 1-5) are used to save dependences for the corresponding tasks in TM0, where each entry can hold 3 dependences. These allow to hold 15 dependences for each task (enough for real applications currently programmed with OmpSs). TM has four actions: Memory Read/Write, New Entry Request (accepts a new entry request and responds with a free entry address) and Finished Entry Request (accepts a finished entry request and recycles this entry from occupied to free). VM has 512 entries and has the same access pattern as TM. We describe DM in Section III-C.

B. Operational Flow

Picos Hardware consists of two major procedures: new and finished task processing in Figure 3b.



(a) Future Architecture

(b) Current Architecture

Fig. 3. Picos Hardware

When a new task arrives (N1-N6):

GW reads its meta-data and dependences (N1). Then it checks for a free TRS slot (TRS number, TM entry). If there is no free slot, GW does not process the new task; otherwise, it obtains one free TRS slot (N2) and dispatches the new task to TRS (N3). After that, if the new task has dependences, GW forwards each of them to DCT (N4).

TRS receives this new task from GW and saves it inside the assigned TM0 slot. If it has no dependences (`#Num.Dep. = #Ready Dep.`), TRS marks it ready and sends it to TS for execution (N6); otherwise, TRS waits for notifications (ready or dependent, N5) for each dependence. For each ready notification, TRS increases the corresponding `#Ready Dep.` by 1 in TM0; and for each dependent packet, TRS saves it in the correct TMX entry. TRS only marks the new task ready after all the dependences are ready.

DCT receives the dependences. For each dependence, DCT checks whether it is dependent on those dependences arrived earlier (N5), and saves it in both DM and VM. If not, DCT sends a ready packet (the corresponding TRS slot and VM address) to TRS; otherwise DCT sends a dependent packet (TRS slot, VM address, dependent TRS slot) to TRS.

When a finished task arrives (F1-F4):

GW reads the finished task (F1), and then distributes it to TRS (F2).

TRS receives the finished task, first checks TM0 for `#Num.Dep.`, checks these `#Num.Dep.` of dependences in TMX. Secondly it sends finished packets (VM address) for each dependence to DCT (F3). Thirdly it deletes the task inside the assigned TM slot (the TM slot now can be recycled as free slot).

DCT receives finish packets of dependences. For each dependence, DCT checks the corresponding VM entry (Consumer, Producer, Count) to see if there are other ones that depend on it. If there exists no such dependences, DCT deletes it from the DM and VM directly; otherwise DCT keeps tracking its consumers/producers, and sends a dependent packet (current TRS slot, VM address) to TRS to wake up the first waiting dependent task (F4). Once all of such dependences that depend on it are resolved and finished, the dependence is deleted inside the VM and DM eventually. Traffic concerning N5/F4 is explained in detail in Section III-D.

C. DM Designs

For each new dependence entering DCT, DM performs address matching against those arrived earlier, to establish data dependences. Later, for each dependence's finish packet from TRS, DM read/write are performed on DM for releasing data dependences. Both processes, finding a slot to save the dependence and address matching, are critical for the prototype performance. On the one hand, each task can have multiple dependences that stresses DCT more than TRS; on the other hand, the whole system may stall if new dependences cannot be stored in DM which can be due to DM conflicts (more than 8 dependences try to use the same DM entry in the case of 8-way) or memory capacity. To overcome possible conflict stalls, three different designs of DM are proposed and evaluated:

- DM 8way, a 64-entry, 8-way associative cache-like memory with direct hash.
- DM 16way, a 64-entry, 16-way associative cache-like memory with direct hash.
- DM P+8way, a 64-entry, 8-way associative cache-like memory with Pearson hashing [20].

Each way in Figure 4 comprises valid (V), input (I), tag, data (VM address, count). Valid indicates if the way is occupied or free; input indicates if all the dependences arrived earlier in this way are input (dependences with the same memory address are saved in the same DM way), thus are independent.

All the three DM designs support three main operations: DM read, write and compare. DM read/write are general memory read/write operations. DM compare operations are

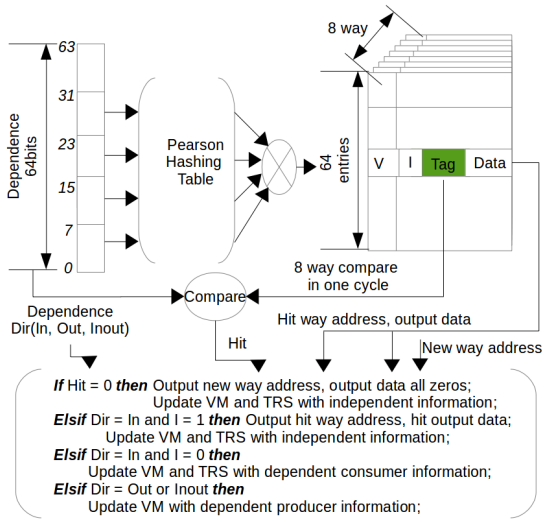


Fig. 4. DM P+8way access diagram

similiar to cache access, and for DM 8/16way versions, the LSB 6 bits of each dependence address are used as index.

For DM P+8way, Figure 4 shows the compare operation. The Pearson hashing function is first applied to each 8 bits of the LSB 32 bits to randomize the value of the dependence address; and then the LSB 6 bits after the xor of these hashing values are used as index to access the 64 entries of memory; one cycle later, the whole 64 bits dependence address is used to compare with the tags of all 8 ways in the entry. Actions following the comparison results are shown in the pseudo code in Figure 4. Note that the New DM address is obtained by checking both Hit and Valid values from all 8 ways. Inside each entry, way 0 has the highest priority, and way 7 has the lowest priority.

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

D. Dependence Chains

It is challenging and crucial to establish dependences and wake up tasks rapidly and economically. Tasks are woken up following different processes depending on the types of their dependences. Figure 5 shows an example of six tasks, where each task has only one dependence (A) with different directions. This example assumes that all the new tasks arrive to Picos before the first one finishes its execution. The six tasks form a mixed Producer-Consumer chain (Task1, 2, 3 and 4) and Producer-Producer chain (Task1, 5, 6) established inside TRS and DCT. The solid line shows how the dependence chain is established according to the sequence of new tasks. The dashed line shows the order in which they are woken up after Task1 finishes. Note that the Producer-Consumer chain is woken up from the last consumer, the Producer-Producer chain is woken up in sequence.

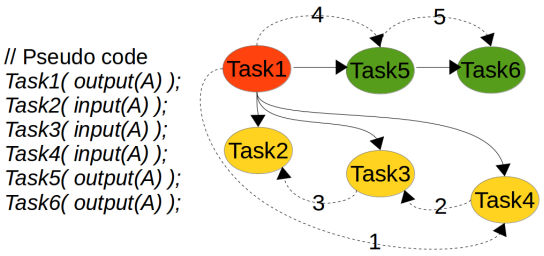


Fig. 5. An example of dependence chain

When Task1 arrives, its dependence is forwarded to DCT. For the first task, its dependence is independent, a new DM and VM entries are assigned to it. The DM entry stores the memory address of the dependence as Tag, and the VM entry stores consumer/producer related information. They also store pointers that point to each other and counters that count the times of appearance of the same dependence that in this case are initialized with 1. For this dependence, as it is independent, DCT sends a ready message to TRS. TRS then saves the VM address of this dependence inside the assigned TMX slot, marks this task ready and sends it to TS for execution.

When Task2 arrives, its dependence is forwarded to DCT. Once DCT receives the dependence, it first does a DM compare and realizes that it is the first consumer of the previous task. In this case, it is saved in the same DM and VM entry (increase the counter of this dependence to 2, and update the consumer TRS slot of Task1 with Task2 in VM).

When Task3 arrives, DCT detects that its dependence is the second consumer. The dependence is then saved in the same DM and VM entry (increase the count of this dependence to 3, and update the Consumer TRS slot of Task2 with Task3 inside VM). At the same time DCT notifies TRS that Task2 will be waken up after Task3. The same happens for Task4 (increase the count of this dependence to 4, and update the Consumer TRS slot section in VM). In this way the last consumer is stored in DCT while the formers are kept chained in TMX slot of the previous task inside the TM. Until now one DM and one VM entry have been assigned.

When Task5 arrives, DCT detects that it is the fifth time when the A dependence appears and it is a producer. A new VM entry is assigned to store this latest version of producer, and the last VM entry is updated to point to this new VM entry. The same happens for Task6 to keep the Producer-Producer chain in DCT. Up to this point, one DM entry and three VM entries have been assigned.

When Task1 finishes, TRS notifies DCT of first A dependence's finish. DCT checks the corresponding VM entry and sends a ready message to TRS for Task4 (link 1 in Figure 5). Once TRS receives this message, it wakes Task4 and sends another ready message (managed by the Arbiter module) to wake Task3 (link 2), and then Task2 (link 3). Now Task2, Task3 and Task4 are marked ready after TRS receives three these ready messages and are sent to execute in idle workers. Whenever a task finishes, TRS notifies DCT. Once DCT receives three finish messages, it wakes up Task5 (link

4) and deletes the first VM entry of the dependence. When Task5 finishes, the process is repeated and the second VM entry is deleted. Finally after Task6 finishes TRS notifies DCT to delete the DM and the third VM entry.

IV. Experimental Setup and Methodology

A. Experimental Setup

We use XILINX ISE Design Suite 14.4, Vivado 14.4, SDK and a Zynq 7000 All-Programmable SoC Platform (Zedboard) to develop the Picos prototype and its embedded system. Zedboard includes one FPGA Chip XC7Z020-CLG484 [21] which comprises a Processing System (PS) (Dual ARM Cortex-A9 MPCore) and a Programmable Logic (PL) part.

The OmpSs programming model in use in this paper for benchmarks is supported by the Mercurium compiler 1.99 and the associated Nanos++ RTS.

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

Sequential and parallel execution time of the same applications from the Picos prototype are obtained in Zedboard by using traces. Traces includes task creation latency in cycles, task identification, dependence address and direction, and task execution time in cycles obtained through instrumenting the sequential execution in the shared machine. Traces are also used to feed a Perfect Simulator which measures critical-path task execution to show the roofline speedup of each OmpSs application. Speedup shown in this paper is computed against the sequential execution time.

B. Hardware-In-the-Loop Simulation Platform

The embedded system also named Hardware-In-the-Loop (HIL) simulation platform is a modern way to validate the functionality and examine the performance of IP-cores. Figure 6 shows the organization of the HIL platform developed for the Picos prototype (simplified for explanation). The PL part uses a 80MHZ global clock, and a 64bits AXI Timer synchronized with the same clock as the global timer. In this subsection, we present two major operational modes of the platform:

HW-only (Solid labeled line): employs a naive process as all the tasks are sent to Picos (1, 2) once, and all the finished tasks are retrieved all at once (5, 6). Workers are implemented inside the PL part so that ready tasks can start executing shortly after there are idle workers (3) and finished tasks are used to notify the Picos for dependence analysis (4).

Full-system (Dashed labeled line): employs a close-loop process. Each task is created and sent to Picos for dependence analysis (1, 2). Each ready task is retrieved from Picos to the

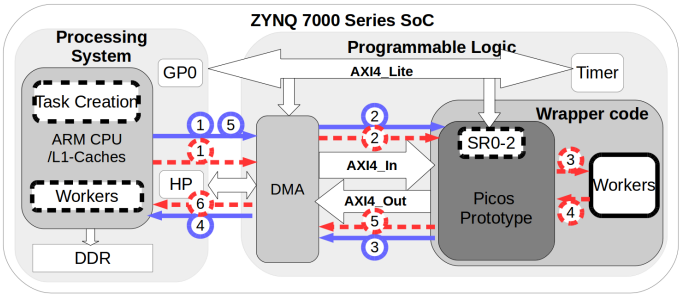
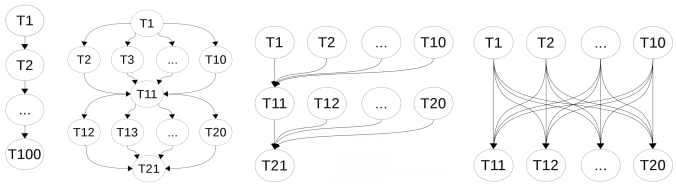


Fig. 6. Hardware-In-the-Loop Platform



(a) C4 (b) C5 (c) C6 (d) C7

Fig. 7. Dependence graphs of synthetic benchmarks

ARM core for execution in the workers (3, 4). Finally each finished task is sent back to notify Picos (5) to carry on the process until the last task. Three queues are employed inside the Picos prototype for new, ready and finished tasks; and SR0-2 are the corresponding status registers. The communication latency for sending or retrieving each task via AXI Stream interface takes around 200 to 300 cycles for each message.

C. Benchmarks

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

Seven synthetic benchmarks are composed of three testcases with independent tasks (Case1-3) and four testcases with dependent tasks (Case4-7). Each testcase has a sequence of 100 tasks, issued every cycle and of length 1 cycle so the processing capacity of the prototype can be measured. Each task in Case1, Case2 and Case3 has 0, 1 and 15 dependences. Figure 7 shows the dependence patterns of the testcases with dependent tasks. Case4 is a single chain of 100 inout dependences; Case5 is 10 sets of 10 consumers for the same producer; Case6 is 10 sets of 10 producers for the same consumer; and Case7 is 10 sets of 10 mixed producers/consumers.

Five real applications Gauss-Seidel Heat, Lu, Sparse Lu, Cholesky [8] and H264dec [22] are selected to study the capability and detect possible bottlenecks in the Picos prototype. Basic functions of these applications are:

- Gauss-Seidel Heat is an iterative Gauss-Seidel solver for heat distribution.
- Lu factorization decomposes an $(m \times n)$ matrix (m>=n) A=LU, with L unit lower triangular $(m \times n)$ and U upper triangular $(n \times n)$.
- Sparse Lu performs a LU decomposition over a square sparse matrix.

TABLE I
REAL BENCHMARKS

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

- Cholesky factorization computes A = LL', with A an $n \times n$ SPD matrix and L lower-triangular.
- H264dec is a high performance H.264 video decoder, a video pedestrian_area.h264 is selected as input.

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

V. RESULT EVALUATION

A. Difference between Picos configurations

To decide the best implementation, we first evaluate the difference of performance with three DM designs. Figure 8 presents their speedup (bar, y-axis), four real benchmarks with a pair of block sizes each are used for testing under HIL HW-only mode.

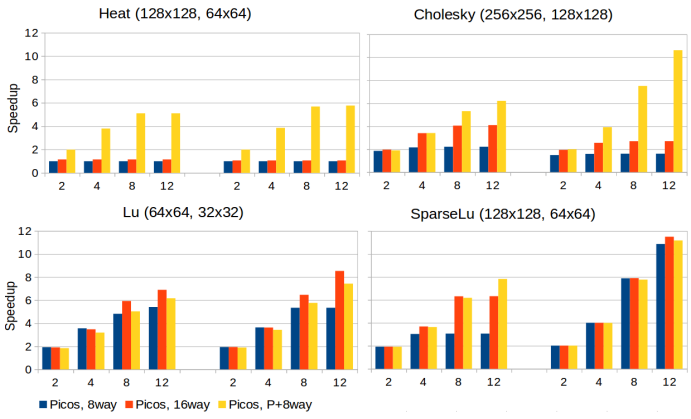


Fig. 8. Speedup: Different Picos configurations

The first row shows the performance of Heat and Cholesky respectively. For both cases, Picos 8way and 16way yield similar results, and the lowest speedups, and both designs do not scale well from 2 to 12 workers, while Picos P+8way achieves the highest speedup and scales relatively well as from 2x to 5.9x with Heat (64x64) and from 2x to 11.5x with Cholesky (128x128).

TABLE II
#DM CONFLICTS IN THREE PICOS DESIGNS

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

The second row shows the performance of Lu and SparseLu. For these two cases, all the three Picos designs benefit from the decreasing block sizes and scales from 2 to 12 workers; in addition, as the number of workers increases, Picos 16way and P+8way yield close to the highest speedup.

To summarize, Picos with DM P+8way yields much better results than the other two designs in most cases. An exception is Lu where Picos with DM 16way achieves better results, we explain this corner case later.

Those performance results greatly depend on the number of DM conflicts. Table II shows the number of DM conflicts detected for benchmarks inside the three Picos designs with 12 workers. Regarding the DM conflicts impact, we can observe that the Picos P+8way is without doubt the best solution with less DM conflicts.

The interesting Lu case is a corner case which is caused due to the way that Picos prototype is designed to awake the Producer-Consumer chain from the last consumer, shown in the operation flow example (Figure 5). With DM P+8way, there are no DM conflicts, so the task dependence graph is created much faster. When the producer task finishes, the

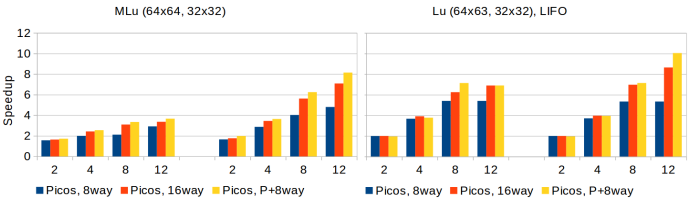


Fig. 9. Performance of Modified Lu

consumer tasks are woken up starting from the last one, causing the schedule of some tasks in the critical path to be postponed and thus resulting in lower speedup. As for DM 16way, the task dependence graph is created much more slowly due to the delays caused by DM conflicts. These tasks in the critical path are therefore scheduled earlier and result in higher speedup. We modified the task creation order of Lu (MLu in Figure 9) to avoid this corner behavior. Results of MLu are shown on the left, from block size 64 to 32, the

Picos with DM P+8way now yields much better results than the others. Note that, different scheduling policies of ready tasks can be used to change this corner case behavior. The results of using a LIFO instead of a FIFO as the TS unit for the original Lu application are shown in Figure 9 right, both figures share the same x-axis.

B. Resource Consumption

Table III shows the resource consumption of both memory and the Picos prototype. The size from DM 8way to 16way is doubled with the objective to speedup this component by using higher associativity to reduce DM conflicts. This significant increase can be observed from the BRAM usage of DM 8way and 16way, 9% to 17% respectively. The corresponding VM is also doubled from 512 to 1024 entries to keep it coherent with the DM size [7].

TABLE III
HARDWARE RESOURCE CONSUMPTION

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

Resource consumption of DM 8way and P+8way are very close (BRAM usage are 9% and 10%), both are much lower than DM 16way. Although the resource consumption of DM 16way is not very demanding, its number of DM conflicts is much higher than of the DM P+8way in Table II. We could have decided to increase the 16way into a 32way doubling the size in order to reduce the DM conflicts, but this would lead to a double increase of the resource usage. Regarding the performance difference in Section V-A and hardware cost, we can conclude that Picos with DM P+8way is the most balanced design among all three. Therefore, in the following section, we will focus on this configuration.

Hardware costs for TRS and DCT are also shown in the table, the other modules GW, TS and ARB are designed simply and their costs are trivial.

C. Latency and Throughput

In this section, the processing capacity (latency and throughput) of the hardware design is evaluated by using the synthetic benchmarks with HIL HW-only mode. After that the influence of integrating the full system (ARM processing, communication and Picos) is analyzed. The cost of integrating hardware and software is mainly composed of two parts: the communication latencies, and the task creation and submission cost of Nanos++ RTS.

In Figure 10, we shows the task creation and submission overhead measured in cycles (y-axis) of Nanos++ RTS with different number of threads (x-axis). `Creation` shows the task creation overhead per task (same for varied number of dependences); `x DEPs` shows task submission overhead for single task with x dependences. For the Nth task, its required overhead is acculmulated based on the N-1 previous tasks.

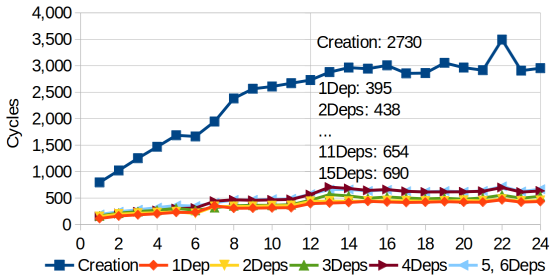


Fig. 10. Nanos++ RTS overhead for single task

TABLE IV
RESULTS OF THE SYNTHETIC BENCHMARKS

Testcase		Independent			Dependent			
		Case1	Case2	Case3	Case4	Case5	Case6	Case7
#d1st/avg#d		0/0	1/1	15/15	1/1	2/2	11/2	11/11
HW-only	L1st	45	73	312	72	96	287	233
	thrTask	15	24	243	24	35	38	178
	thrDep	-	24	16	24	18	19	16
HW+comm.	L1st	1172	1174	1293	1151	1158	1274	1279
	thrTask	740	740	734	743	743	743	743
	thrDep	-	740	49	743	371	372	68
Full-system	L1st	3879	4240	4710	4246	4217	4531	4549
	thrTask	2729	3125	3413	3124	3168	3165	3379
	thrDep	-	3125	228	3124	1584	1583	307

Table IV shows the processing capacity of Picos P+8way with HIL HW-only, HW+communication and Full-system modes, 12 workers are used. HIL HW+communication adds communication latencies based on the HW-only, and no task creation and submission cost are considered. Row (#d1st/avg#d) indicates the number of dependences for the first task and the average number of them for all tasks. For each testing mode, three other aspects are evaluated in cycles.

First, in HIL HW-only mode, the latency of the 1st task (`L1st`) is proportional to the number of dependences of the lst task (`#d1st`), while the increasing degree slows down as the average number of dependences in the testcase (`avg#d`) increases. From Case1/4 to Case3/7, `L1st` increases from 45/72 to 312/233 cycles. From Case1 to Case2, it increases 24 cycles while from Case2 to Case3 it only increases 16 cycles for each dependence. This can be seen as the latency and repetition rate of the first and following independent instructions flow into a pipelined funtional unit.

Second, throughput for additional tasks (`thrTask`) mainly depends on average number of dependences (`avg#d`) and the type of the dependences. For tasks with the same average number of dependences as Case2 and Case4, the `thrTask` are similar. While for Case6 to Case7 with 2 and 11 dependences respectively, it increases from around 38 to 178 cycles. And from Case5 to Case6, there is a minor increase due to dependence difference. However, this effect in the HW-only mode is nearly hidden when the communication and Nanos

overheads are introduced, showing that the hardware part is fast enough with the current integration.

Third, throughput for additional dependences (`thrDep`) remains stable in all the testcases and decreases when the average number of dependences (`avg#d`) increases. For cases with more than one dependence, the throughput per dependence is 16 to 19 cycles. For Case2 and Case4 which have one dependence, it is 24 cycles. Same pipelined influence can be explained for this as for `thrTask`. This effect in the HW-only is greatly enhanced in the Full-system mode where the communication and Nanos overheads become the main performance factor. As can be seen, from HIL HW+communication to Full-system, the `thrDep` from Case2 to Case3 drops from 740 to 49 cycles and from 3125 to 228 cycles.

Regarding the Full-system mode, as the number of dependences increases, the time required to process a task `thrTask` remains stable while the `thrDep` decreases proportionally. This is a key factor contributing to the performance of the Picos HIL Full-system presented which makes the Picos alternative not only more powerful but also steadier, as this effect doesn't appear in the software-only implementation.

D. Scalability

Finally, to show the real potential of the Picos prototype, in this section, five real benchmarks with varied block sizes are used to study their scalability under HIL Full-system mode with up to 24 workers. Figure 11a to 11e show the speedup (y-axis) of Heat, SparseLu, Lu, Cholesky and H264dec with four block sizes obtained by Picos under HIL Full-system, Perfect Simulator and Nanos++ RTS.

Results of Perfect Simulator shows the available parallelism peak for these applications. For Heat in Figure 11a, SparseLu in Figure 11d, Lu in Figure 11c and Cholesky in Figure 11b, the Picos prototype achieves nearly roofline speedup with block sizes from 256 to 64 and with the number of workers from 2 to 24. For H264dec in Figure 11e, the Picos prototype scales well with block size (8, 4, 2, 1) and up to 12 cores, then remains stable.

For Heat in Figure 11a, Cholesky in Figure 11b with block size 32 and H264dec in Figure 11e, there are emerging gaps between the results obtained by the Picos prototype and Perfect Simulator. The reason is that Picos in use is the simplest configuration which is unable to unfold such a high parallelism from applications here, also due to the lack of hardware resources to manage so many processors. The Picos prototype with more module instances should be able to obtain higher speedup and fill this gap [7].

For all the five real benchmarks, there are two main observations. First, for each benchmark with a fixed block size, Nanos++ RTS scales up to 8 workers maximum while the Picos prototype continues to scale to 24 workers in some cases. For example, for SparseLu in Figure 11d with block size 32 and Cholesky in Figure 11b with block size 64, the Picos prototype achieves 16x to 24x and 15x to 21x with 16 to 24 workers, respectively.

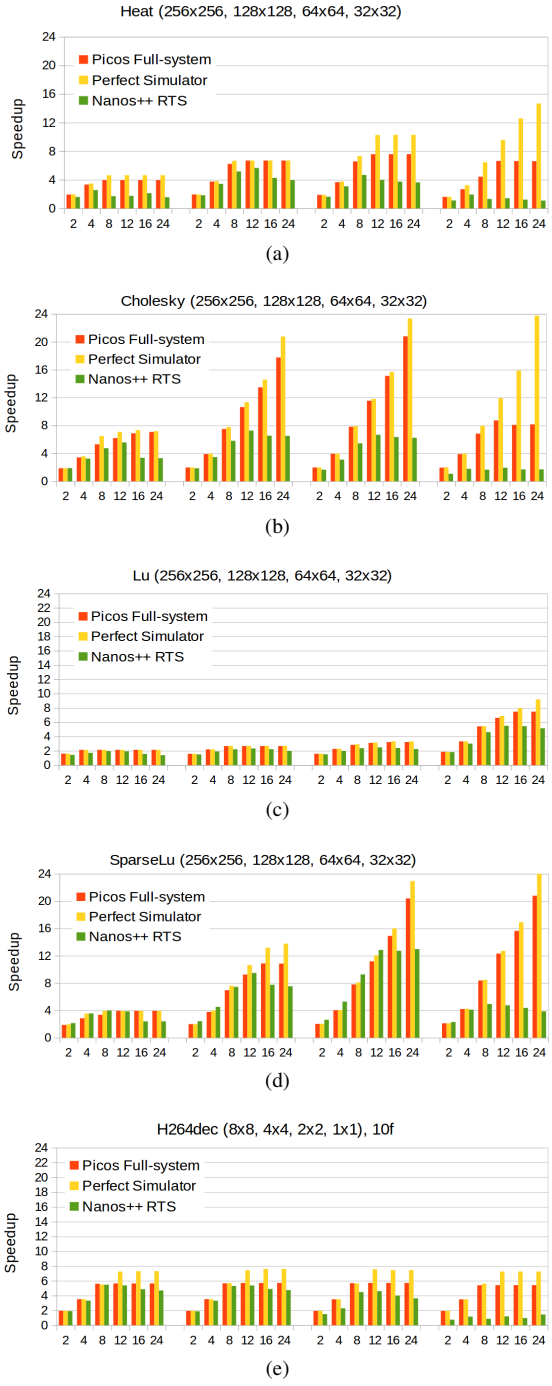


Fig. 11. Scalability study of real benchmarks

Secondly, for each benchmark, as its block size decreases, Nanos++ RTS starts to degrade rapidly after some point while the Picos prototype keeps on advancing or at least remains stable. For example, in Figure 11a, the speedup achieved by Nanos++ RTS for Heat (64 to 32) drops from 4.5x to 1.6x with 8 workers while the speedup of the Picos prototype remains stable as 6.3x; and for SparseLu in Figure 11d and Lu in Figure 11c with block size 64 to 32, when the speedup achieved by Nanos++ RTS starts to degrade, the Picos prototype continues to advance from 3.3x to 8x with

16 workers. In Figure 11e, for H264dec, as the block size and the performance of Nanos++ RTS decreases, the Picos prototype's performance remains stable. All of the above prove that even the simplest configuration of Picos Hardware fulfills our expectation, while larger configurations are expected to be able to cope with future manycores.

VI. Main Lessons

We learnt important lessons by building the Picos prototype. First, using limited hardware resources should be carefully compensated to overcome potential deadlocks. We decouple the task and dependence management to enable a balance between hardware cost and the complexity. We also design three different DMs to reduce DM conflicts and avoid memory capacity stalls. These are key to design a high performance accelerator with low cost. Secondly, the way that data is exchanged between the processors and the hardware accelerator is very important, as the transfer overhead might overcome the dependence management time. Finally, there are gaps between the C-simulation model and final hardware. To maintain the performance, it is crucial to adapt your design methodology from SW-friendly to HW-friendly.

VII. Conclusion

In this paper we present the very first hardware prototype of Picos, as a RTS hardware support to speedup the task and dependence management for task-based dataflow programming models like OpenMP 4.0 and OmpSs. The presented implementation has been fully analyzed and tested on a real embedded system on a Zynq 7000 All-programmable SoC Platform.

The prototype is able to manage up to 256 in-flight tasks with up to 15 dependences each. Design exploration of different designs focused on the trade-offs between task and dependence management are implemented and evaluated. One of the designs - an 8 way associative cache-like structure with Pearson Hashing - achieves better performance over the others while keeping the hardware consumption at bay. Scalability studies are performed on the prototype with the optimum configuration using a set of real benchmarks - Gauss-Seidel Heat, Lu, SparseLu, Cholesky and H264dec. Results show that the prototype greatly outperforms the existing OmpSs software-only implementation (Nanos++) and as the task granularity decreases, the prototype continues to scale after Nanos++ RTS starts to degrade. More importantly, with a larger design with multiple task and dependence management units upcoming, Picos Hardware could be able to exploit a larger magnitude of parallelism in the applications with very fine granularity, that software alternatives cannot achieve.

Acknowledgment

This work is supported by the Spanish Government through Programa Severo Ochoa (SEV-2011-0067), by the Spanish Ministry of Science and Technology through TIN2012-34557 project, by the Generalitat de Catalunya (contract 2009-SGR-980) and by the European Research Council under the European Unions 7th FP, ERC Grant Agreement number 321253. We also thank the Xilinx University Program for its hardware and software donations.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *6th Symposium on Operating Systems Design & Implementation*, 2004.

[2] J. Reinders, *Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism*. OReilly Associates, 2007.

[3] O. ARB, "Openmp application program interface, v4.0." [online], 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[4] J. M. Perez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architecture," in *International Conference on Cluster Computing(CC)*, 2008.

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

[6] B. S. Center, "Ompss user guide." [online], 2015. http://pm.bsc.es/ompss-docs/user-guide/OmpSsUserGuide.pdf.

[7] F. Yazdanpanah, C. Alvarez, D. Jimenez-Gonzalez, R. M. Badia, and M. Valero, "Picos: A hardware runtime architecture support for ompss," *Future Generation Computer Systems(FGCS)*, 2015.

[8] B. S. Center, "Bsc application repository(bar)." [online], 2014. https://pm.bsc.es/projects/bar/wiki/Applications.

[9] N. Engelhardt, T. Dallou, A. Elhossini, and B. Juurlink, "An integrated hardware-software approach to task graph management," in *In 16th IEEE International Conference on High Performance and Communications(HPCC-2014)*, 2014.

[10] 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 *IEEE / ACM International Symposium on Microarchitecture (MICRO-43)*, 2010.

[11] F. Yazdanpanaha, D. Jimenez-Gonzalez, C. Alvarez-Martinez, Y. Etsion, and R. M. Badia, "Analysis of the task superscalar architecture hardware design," in *International Conference on Computational Science (ICCS)*, 2013.

[12] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *International Symposium on Computer Architecture*, 2007.

[13] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2010.

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

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

[16] M. Sjalander, A. Terechko, and M. Duranton, "A look-ahead task management unit for embeded multi-core architectures," in *Conference on Digital System Design(DSD)*, 2008.

[17] D. Capalija and T. S. Abdelrahman, "Microarchitecture of a coarse-grain out-of-order superscalar processor," in *International Transaction on Parallel and Distributed Systems*, 2013.

[18] T. Dallou, A. Elhossini, B. Juurlink, and N. Engelhardt, "Nexus#: A distributed hardware task manager for task-based programming models," in *2015 IEEE 29th International Parallel and Distributed Processing Symp(IPDPS)*, 2015.

[19] M. C. Jefferey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *Proceedings of the 48th International Symposium on Microarchitecture(ACM)*, 2015.

[20] P. K. Pearson, "Fast hashing of variable-length text strings," in *Communication of the ACM*, 1990.

[21] XILINX, "Zynq-7000, etc.." [online], 2015. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[22] TU-Berlin, "Starbench benchmark suite." [online], 2015. http://www.aes.tu-berlin.de/menue/forschung/projekte/abgeschlossene_projekte/starbench_parallel_benchmark_suite/.