Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 00 (2013) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2013

Analysis of the Task Superscalar Architecture Hardware Design

Fahimeh Yazdanpanah[a,b], Daniel Jimenez-Gonzalez[a,b], Carlos Alvarez-Martinez[a,b], Yoav Etsion[c], Rosa M. Badia[a,b]

[a] *Universitat Politècnica de Catalunya (UPC), Barcelona 08034, Spain*
[b] *Barcelona Supercomputing Center (BSC), Barcelona 08034, Spain*
[c] *Technion – Israel Institute of Technology, Haifa 32000, Israel*

Abstract

In this paper, we analyze the operational flow of two hardware implementations of the Task Superscalar architecture. The Task Superscalar is an experimental task based dataflow scheduler that dynamically detects inter-task data dependencies, identifies task-level parallelism, and executes tasks in the out-of-order manner. In this paper, we present a base implementation of the Task Superscalar architecture, as well as a new design with improved performance. We study the behavior of processing some dependent and non-dependent tasks with both base and improved hardware designs and present the simulation results compared with the results of the runtime implementation.

Keywords:
Task Superscalar; Hardware task scheduler; VHDL

1. Introduction

Concurrent execution assumes that each discrete part of a program, called task, is serially executed in a processor and that the execution of multiple parts appears to happen simultaneously. However, exploiting concurrency (i.e., creating, managing synchronization of tasks) to achieve greater performance is a difficult and important challenge for current high performance systems. Although the theory is plain, the complexity of traditional parallel programming models in most cases impedes the programmer to harvest performance.

Several partitioning granularities have been proposed to better exploit concurrency. Different dynamic software task management systems, such as task-based dataflow programming models [1, 2, 3, 4, 5], benefit dataflow principles to improve task-level parallelism and overcome the limitations of static task management systems. These models implicitly schedule computation and data and use tasks instead of instructions as a basic work unit, thereby relieving the programmer of explicitly managing parallelism. While these programming models share conceptual similarities with the well-known Out-of-Order superscalar pipelines (e.g., dynamic data dependency analysis and dataflow scheduling), they rely on software-based dependency analysis, which is inherently slow, and limits their scalability. The aforementioned problem increases with the number of available cores. In order to keep all the cores busy and accelerate the overall application performance, it becomes necessary to partition it into more and smaller tasks. The task scheduling (i.e., creation and management of the execution of tasks) in software

*Corresponding author. Tel.: +34 93 401 16 51; fax: +34 93 401 0055.
E-mail address: {fahimeh, djimenez, calvarez}@ac.upc.edu, yetsion@tce.technion.ac.il, rosa.m.badia@bsc.es.

introduces overheads, and so becomes increasingly inefficient with the number of cores. In contrast, a hardware scheduling solution can achieve greater speed-ups as a hardware task scheduler requires fewer cycles than the software version to dispatch a task. Moreover, a tiled hardware task scheduler is more scalable and parallel than the equivalent software.

The Task Superscalar [6, 7] is a hybrid dataflow/von-Neumann architecture that exploits task level parallelism of the program. Therefore, the Task Superscalar combines the effectiveness of Out-of-Order processors together with the task abstraction, and thereby provides a unified management layer for CMPs which effectively employs processors as functional units. The Task Superscalar has been implemented in software with limited parallelism and high memory consumption due to the nature of the software implementation. A hardware implementation will increase its speed and parallelism, reducing the power consumption at the same time. In our previous work [8], we presented the details of designing the different modules of a base prototype of hardware implementation of the Task Superscalar architecture. In this paper, we analyze of the hardware implementation of the Task Superscalar architecture. Our analysis is based on the base prototype of hardware implementation of the Task Superscalar architecture [8] and an improved design that is presented in this paper. Using some testbenches, we simulate and evaluate the hardware designs of the Task Superscalar architectures, and compare the results to the ones obtained by running the same test cases on the Nanos runtime system [9] supports the same programming model.

The reminder of the paper is organized as follows: In Section 2, we present the related work and a brief overview of the Task Superscalar architecture. Section 3 describes the hardware designs of the Task Superscalar architecture and the operational flow of the pipeline of the improved hardware design. In Section 4, we introduce our experimental setup and methodology, and, then, results and evaluation of hardware designs are presented in this Section. Finally, the paper concludes in Section 5.

2. Related Work

An emerging class of task-based dataflow programming models such as StarSs [1, 2, 3], OoOJava [5] or JADE [4] automates data dependency and solves the synchronization problem of static task management systems. These models try to support dynamic task management (creation and scheduling) with a simple programming model [1]. However, their flexibility comes at the cost of a rather laborious task management that should be done at runtime [10]. Moreover, management of a large amount of tasks affects the scalability and performance of such systems and potentially limits their applicability.

Some hardware support solutions have been proposed to speed-up task management, such as Carbon [11], TriMedia-based multi-core system [12] and TMU [13], but most of them only schedule independent tasks. In these systems, the programmer is responsible to deliver tasks at the appropriate time. Carbon minimizes task queuing overhead by implementing task queue operations and scheduling in hardware to support fast tasks dispatch and stealing. TriMedia-based multi-core system contains a centralized task scheduling unit based on Carbon. TMU is a look-ahead task management unit for reducing the task retrieval latency that accelerates task creation and synchronization in hardware similar to video-oriented task schedulers [14].

Dynamic scheduling for system-on-chip (SoC) with dynamically reconfigurable architectures is interesting for the emerging range of applications with dynamic behavior. As an instance, Noguera and Badia [15, 16] presented a micro-architecture support for dynamic scheduling of tasks to several reconfigurable units using a hardware-based multitasking support unit. In this work the task dependency graph is statically defined and initialized before the execution of the tasks of an application.

Task Superscalar architecture [6, 7] has been designed as a hardware support for the StarSs programming model. Unlike in Noguera's work, the task dependency graph is dynamically created and maintained using runtime data flow information, therefore increasing the range of applications that can be parallelized. The Task Superscalar architecture provides coarse-grain managed parallelism through a dynamic dataflow execution model and supports imperative programming on large-scale CMPs without any fundamental changes to the micro-architecture. As our work is based on Task Superscalar architecture, in the following section we describe this architecture. Nexus++ [17, 18] is another hardware task management system designed based on StarSs that is implemented in a basic SystemC simulator. Both designs leverage the work of dynamically scheduling tasks with a real-time data dependence analysis while, at the same time, maintain the programmability, generality and easiness of use of the programming model.

2.1. Task Superscalar Architecture

The Task Superscalar architecture uses the StarSs programming model to uncover task level parallelism. This programming model enables programmers to explicitly expose task side-effects by annotating the directions of the parameters. With those annotations, the compiler can generate the runtime calls that will allow the Task Superscalar pipeline to dynamically detect inter-task data dependencies, identify task-level parallelism, and execute tasks in the out-of-order manner.

Figure 1 presents the organization of the Task Superscalar architecture. A task generator thread sends tasks to the pipeline front-end for data dependency analysis. The recently arrived tasks are maintained in the pipeline front-end. The front-end asynchronously decodes the task dependencies, generates the data dependency graph, and schedules tasks when all their parameters are available (i.e., following dataflow philosophy). Ready tasks are sent to the backend for execution. The backend consists of a task scheduler queuing system (TSQS) and processors.

The Task Superscalar front-end employs a tiled design, as shown in Figure 1, and is composed of four different modules: pipeline gateway (GW), task reservation stations (TRS), object renaming tables (ORT) and object versioning tables (OVT). The front-end is managed by an asynchronous point-to-point protocol. The GW is responsible for allocating TRS space for new tasks, distributing tasks and their parameter to the different modules, and blocking the task generator thread whenever the pipeline fills. TRSs store the meta-data of the in-flight tasks and, for each task, check the readiness of its parameters. TRSs maintain the data dependency graph, communicating with each other in order to relate consumers to producers and notify consumers when data is ready. The ORTs are responsible to match memory parameters to the most recent task accessing them, and thereby detect object dependencies. The OVTs save and control all the live versions of every parameter in order to manage dependencies. That helps Task Superscalar to maintain the data dependency graph as a producer-consumer chain. The functionality of the OVTs is similar to a physical register file, but only to maintain meta-data of parameters.

Figure 1 also shows at its right the Out-of-Order equivalent component to the Task Superscalar modules. The Task Superscalar extends dynamic dependency analysis in a similar way to the traditional Out-of-Order pipeline, in order to execute tasks in parallel. In Out-of-Order processors, dynamic data dependencies are processed by matching each input register of a newly fetched instruction (i.e., a data consumer), with the most recent instruction that writes data to that register (data producer). The instruction is then sent to a reservation station to wait until all its input parameters become available. Therefore, the reservation stations effectively store the instruction dependency graph composed by all in-flight instructions.

The designers of the Task Superscalar pipeline opted for a distributed structure that, through careful protocol design that ubiquitously employ explicit data accesses, practically eliminates the need for associative lookups. The benefit of this distributed design is that it facilitates high levels of concurrency in the construction of the dataflow graph. These levels of concurrency trade off the basic latency associated with adding a new node to the graph with overall throughput. Consequently, the rate in which nodes are added to the graph enables high task dispatch throughput, which is essential for utilizing large many-core fabrics.

In addition, the dispatch throughput requirements imposed on the Task Superscalar pipeline are further relaxed by the use of tasks, or von-Neumann code segments, as the basic execution unit. The longer execution time of

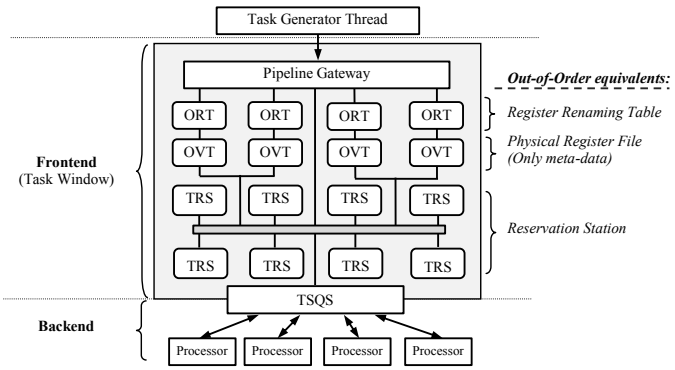

Fig. 1. The Task Superscalar architecture

tasks compared to that of instructions means that every dispatch operation occupies an execution unit for a few dozen microseconds, and thereby further amplifies the design's scalability.

3. Hardware Prototypes of the Task Superscalar Architecture

In our previous work [8], we have presented the detail of each module of the base hardware prototype of the Task Superscalar architecture. Each module is written in VHDL and synthesized into two FPGAs of Virtex 7 family. In this work, we focus on the behavior of the whole Task Superscalar implementation using these modules. Figure 2-a shows the structure of the base design that includes one GW, two TRSs, one ORT, one OVT and one TSQS. These modules communicate with each other using messages (packets).

Figure 2-b illustrates an improved version of the base design. In the new design, we have merged the ORT and the OVT in order to save hardware resources and reduce the latency for processing both new and finished tasks. The new component is called *extended ORT* (eORT). As Figure 2-b shows, the modified design is mainly composed of one GW, two TRSs, one eORT, and one TSQS.

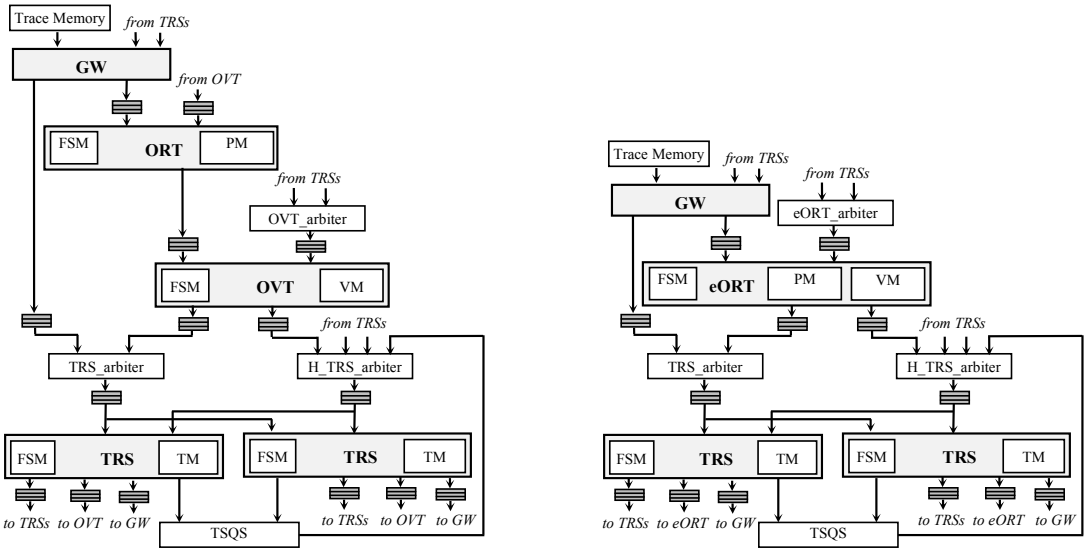

Fig. 2. Task Superscalar hardware designs, a) base prototype, b) improved prototype

One of the objectives in the hardware prototype implementation is to minimize the FPGA resources used in controllers, buses, and registers to maximize the available FPGA resources for the memory units of the modules. TRS memory (TM) is divided into slots. Each slot has sixteen 200-bit entries, one for every meta-data of the task. With this size, each TRS can store up to 512 tasks, so it has a total of 8K entries of memory organized as 512 sixteen-entry slots (one slot per task). Therefore, the whole prototype with two TRSs can store up to 1024 in-flight tasks. The memory of the OVT, called *versions memory* (VM), can save up to 8K versions of parameters. The ORT memory, called *parameters memory*, (PM) has 1024 sets and each set has eight ways. It can store the meta-data of 8K parameters. In the improved design, the eORT has two memories, one for saving parameters (PM) and the other for saving the versions (VM). The PM is an 8-way set-associative memory with 1024 sets for storing the meta-data of 8K parameters. The VM is a direct mapped memory that can save up to 8K versions of parameters.

Another important consideration was to minimize the cycles required for processing input packets in order to increase the overall system speed. Each module of the prototype has a finite state machine (FSM) in order to process the main functionality of the module. In that design, the FSMs are implemented in such a manner that each state uses one clock cycle.

As Figure 2 illustrates, we have used four-element FIFOs and arbiters to interconnect the different modules of the prototype. The FIFOs decouple the processing of every component in the system. This interconnection organization allows the system to scale while reduce the possibility of stalls.

For designing these prototypes, we have modified the operational flow of processing arrived and finished tasks of the original Task Superscalar. Therefore, the hardware design has fewer packets that are also denser than the packets used in the software design. For instance, for sending non-scalar parameters, we have modified the information flow, so we have removed two packets that are used in the software version. In software version, non-scalar parameters are sent to the ORT. After that, the ORT sends a request for creating a version to the OVT and meanwhile, passes the parameter to the TRS. Then, the TRS asks the OVT for the address of the version in the memory of the OVT. After processing the request for creating a version, the OVT informs the address of the version to the TRS. In contrast, in the hardware version, instead of the ORT, the OVT is responsible for sending both the parameter and address of the version to the TRS, after processing the request for creating a version. With these modifications the functionality is maintained (without parameter renaming) diminishing the time needed to process a task.

In addition, since in the hardware design, the allocating and deleting of VM entries is controlled by the ORT, we have removed another packet which was originally sent from the ORT to the OVT as a response for asking permission for releasing a version. Moreover, in the hardware designs, for creating producer-consumer chains, fewer packets are used. Therefore, we have less traffic between modules and also fewer cycles for creating the chains.

3.1. Operational Flow of the Improved Prototype

In this section, we describe the operation flow of the improved design for processing arrived and finished tasks.

Algorithm 1 shows the operational flow of processing arrived tasks that begins when the GW sends an allocation request to one of the TRSs. The GW gets a task from the task generator thread and selects a TRS that has space for saving the task. After selecting a TRS, the GW sends a request to the TRS to allocate a slot of TM for meta-data of the task. Once a slot is allocated, the GW starts to send the scalar parameters of the allocated task to the TRS. For data dependency decoding, the GW sends non-scalar parameters to the eORT. When all of the parameters of a task are sent, the GW is ready to send allocation request of the next task.

When the eORT receives a parameter from the GW, it checks the existence of any entry for this parameter. If there is an entry for the parameter in the PM, the eORT updates it; otherwise, a new entry is created. For every output parameter (producer), eORT creates a new version in the VM for that and updates the previous version of the parameter, if it exists. For every input parameter, if it is the first time that the parameter appears, the eORT creates a new version for it. Otherwise, it only updates the existing version of the parameter adding a new consumer. Meanwhile, the eORT sends the parameter with version information to the TRS.

Algorithm 2 shows the procedure of a finished task. When execution of a task finishes, the TRS starts to release the parameters of the task and also notifies the eORT releasing of each parameter. After all the parameters of the task are released, the TRS frees the task.

For each output parameter, if there are consumers waiting for its readiness, the eORT notifies the TRS that is on the top of the consumer stack that the parameter is ready. When a consumer TRS gets a *Ready* message for a parameter, it updates the associated memory entry and, if there is another TRS consumer, sends the message to it. The *ready* message is propagated between producer and consumers based on consumer chaining that is repeated until there are no more consumers in the stack. In this architecture, each producer does not send a ready message of an output parameter to all of its consumers; instead, a producer sends a ready message only to one of the consumer which is on top of the consumer stack of that parameter [6]. Then each consumer of an output parameter passes the *ready* message to the next consumer. When execution of all of consumers of a producer (an output parameter) finished, this producer sends a *ready* message to the next producer of that parameter.

Meanwhile, when the eORT receives notification of releasing a parameter, it decreases the total number of users of that parameter. If there are no more users for the version it may be deleted. In the case that the version is the last, and there are no more users for the parameter, the eORT entry and its related version in the VM are deleted. In the case that the version is not the last one and there are no more users for the parameter, the eORT frees two entries of the VM (one for the version that will not be used more, and the other for the last version) and also deletes the corresponding eORT entry. The reason of this behavior is the fact that the last version of a parameter should not be deleted before all previous versions are deleted because if a new consumer arrives, it should be linked to the last version.

Although the above description focuses on the decoding of individual tasks and parameters, the pipeline performance stems from its concurrency. As the GW asynchronously pushes parameters to the eORT, the different decoding flows, task executions and task terminations occur all in parallel.

Algorithm 1: Procedure of processing an arrived task

```
GW gets meta-data of a task and its parameters from trace memory;
GW selects a free TRS based on the round robin algorithm;
GW sends the task to the allocated TRS;
if #param = 0 then
    TRS sends the task for execution;
else
    for all parameters do
        if parameter is scalar then
            GW directly sends the parameter to the TRS;
            TRS saves it in the TM;
            if all the parameters are ready then
                TRS sends the task for executing;
        else
            GW sends each non-scalar parameter to eORT for data dependency analysis;
            eORT saves the parameter in PM;
            if parameter is input then
                if first time then
                    eORT creates a version for the parameters in the VM;
                else
                    eORT updates the current version of the parameter in the VM;
            else
                eORT creates a version for the parameter in the VM;
                if NOT first time then
                    eORT updates the previous version of the parameter in the VM;
            eORT sends the parameter to the TRS;
            TRS saves it in the TM;
            if all the parameters are ready then
                TRS sends the task for executing;
```

Algorithm 2: Procedure of processing a finished task

```
TRS releasing all parameters and task from the memory;
TRS notifies the eORT for each output parameter;
if parameter is output then
    eORT notifies readiness of the parameter to the TRS which is the top element of the consumer stack;
if #users of the version = 0 then
    if the version is the last one then
        if all other version deleted then
            eORT deletes the last version and the eORT entry;
    else
        eORT deletes the version;
        if all other version deleted then
            eORT deletes the last version and the eORT entry;
```

4. Results and Evaluation

In this section we detail the results for the base and the improved hardware prototypes. The objectives are: (1) to analyze the latency of managing one isolated task with different number and types of parameters, and (2) compare a real software runtime systems and our prototype proposals. In particular, we analyze 14 different cases

that represent the best, the worst and the average cases for isolated tasks, and then, the influence of the data dependency management of several tasks.

4.1. Experimental Setup and Methodology

The hardware prototypes have been written in VHDL. To verify the functionality of the pipeline of the designs, we have simulated them using the waveform feature of the ModelSim 6.6d. In this context, we use different bit-stream test-benches as individual tasks and sets of non-dependent and dependent tasks.

For the real software runtime system results, we use the experimental results for the Nanos 0.7 runtime system, that supports the OmpSs programming model [19]. We have coded in C the same examples of non-dependent and dependent tasks used for our prototypes experiments, using the OmpSs directives to specify the input and output dependencies. Those codes have been compiled with the Mercurium 1.3.5.8 source to source compiler [9], that has Nanos 0.7a support and uses, as backend compiler, the gcc 4.6.3 (with -O3 optimization flag). The compilation has been done with Extrae 2.3 instrumentation linking options. We have run the applications in a 2.4GHz Core2 Duo machine, with OMP_NUM_THREADS=2 and Extrae instrumentation options. Each application execution generates a trace that is translated to a Paraver trace, which has been analyzed to obtain the execution time spent in the runtime library on managing the tasks and data dependency tasks.

4.2. Hardware Analysis

Table 1 illustrates the latency cycles required for processing isolated tasks with different number of parameters and status of the parameter in the base and improved hardware prototypes. We have selected these cases to find out the minimum and maximum cycles that required for processing different kinds of tasks: tasks without parameter, tasks with one parameter, tasks with two parameters, and tasks with fifteen (i.e., maximum number of parameter in this prototype) parameters. As the Table shows, the minimum latency cycles of processing parameters are for scalar parameters which are the same in both designs. The maximum latency cycles are for processing output (or inout) parameters that do not appear for the first time in the pipeline. The results show that the improved version (with the eORT: OVT + ORT) has fewer latency cycles than the base prototype.

Table 1. Latency cycles for processing isolated tasks

	#params	Condition of the parameter(s)	Latency for processing tasks (cycles)	
			Basic design	Improved design
Case 1	0	-	17	17
Case 2	1	scalar	32	32
Case 3	1	non-scalar (input or output) and first time	50	43
Case 4	1	non-scalar, input, and not first time	51	44
Case 5	1	non-scalar, output, and not first time	52	45
Case 6	2	both scalar parameters	44	44
Case 7	2	1st: scalar 2nd: non-scalar (output or input) and first time	55	48
Case 8	2	1st: scalar 2nd: non-scalar, input and not first time	56	49
Case 9	2	1st: scalar 2nd: non-scalar, output and not first time	57	50
Case 10	2	1st: non-scalar, input and first time 2nd: non-scalar, output and first time	64	51
Case 11	2	1st: non-scalar, input and not first time 2nd: non-scalar, output and not first time	67	54
Case 12	2	both non-scalar, output and not first time	68	56
Case 13	15	all scalar	148	148
Case 14	15	all non-scalar, output and not first time	198	186

Figures 3 and 4 show examples of non-dependent and dependent tasks. In these examples, we have five tasks (`Ti`), each of them has two parameters. Each `Si` is related to the task `Ti` and indicates a TRS_entry (a slot of TM) which is assigned to the `Ti`. TRSs are assigned to the tasks according to the round robin algorithm, so in this examples `T1`, `T3`, and `T5` are stored in one TRS of the prototypes and `T2` and `T4` are stored in the other TRS.

Figure 3 shows the slots and versions of five non-dependent tasks. In the Figure, `Vxi` is the corresponding version of the parameter `xi` that stored in the VM. For the base prototype, it takes 163 cycles for completing this trace while the improved design requires 147 cycles to process these tasks. The hardware designs operate at about 150 MHz.

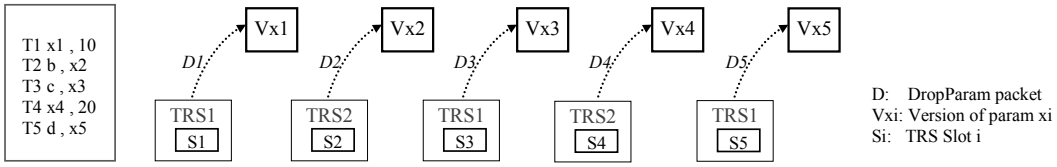

Fig. 3. An example of five non-dependent tasks.

Figure 4 presents an example of a set of five dependent tasks. By this example, we show the producer-consumer chain of hardware designs. The producer-consumer chain of the parameter `x` is shown in Figure 4. `Vix` is a version of `x` in the VM. Task `T1` is a producer for `x` and `T2` and `T3` are its consumers. Therefore, tasks `T2` and `T3` are dependent on `T1`. `V1x` is the corresponding version for these three tasks. `T4` is another producer for `x` and `T5` is its consumer. For these two tasks, we have version `V2x` in the VM.

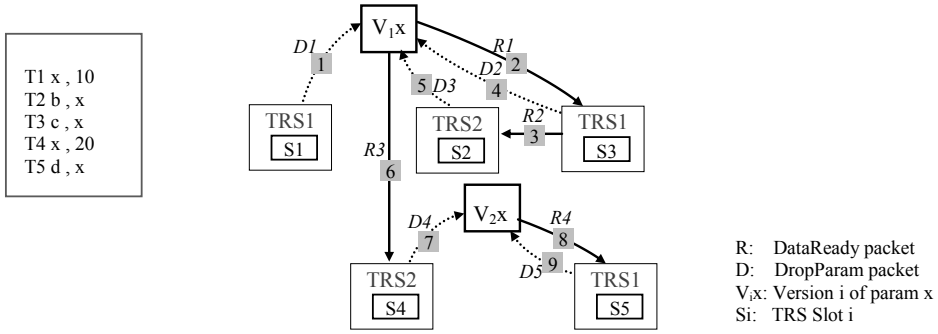

Fig. 4. An example of five dependent tasks

When `T1` finished, the TRS which is responsible for `T1` sends a message (called *DropParam* packet) to its related version (`V1x`) in order to notify it releasing parameter `x`. Then, `V1x` sends a ready message (`R1`) to notify the readiness of `x` to `S3` which is on the top of the consumer stack. `S3` immediately forwards the ready message to `S2` which is another (and also the last) consumer of `x` (produced by `T1`). Whenever `T2` and `T3` finished, `S2` and `S3` inform the `V1x`. As soon as all the consumers related to `V1x` finished, `V1x` sends a ready message to the TRS which saves the next producer of `x` (i.e., `T4`). A similar scenario is done for `T4`, `V2x` and `T5`. For the base prototype, it takes 212 cycles for completing this trace while the improved design spends 195 cycles. Operating frequency for both designs is near 150 MHz.

Table 2 shows the overall number of cycles and time that those two examples take on the two prototypes: base and improved design. The table also shows the latency in cycles and time for those examples on the real software runtime system Nanos. The execution time/cycles of the tasks is the same in both designs and Nanos runtime and depends on the kind of backend processors. Therefore, we have decided to not include their latency/time in the overall count. The table also shows the task throughput (tasks executed per second) for the hardware designs and runtime implementation. Our designs are more than 100x faster than the software runtime.

Figure 5 shows the processing of the packets related to the tasks, their parameters and their dependencies for the two testbenches on our two prototypes (The description of the labels is presented in the caption). In particular,

Table 2. Latencies of processing five tasks on the prototypes and Nanos

	Latency for processing tasks (cycles)			Latency for processing tasks (µs)			Task Throughput (task/ sec)		
	Nanos runtime (2.40 GHz)	Base design (150 MHz)	Improved design (150 MHz)	Nanos runtime	Base design	Improved design	Nanos runtime	Base design	Improved design
Example 1: 5 non-dependent tasks	413×10^3	163	147	172	1.087	0.98	30×10^3	4600×10^3	5100×10^3
Example 2: 5 dependent tasks	475×10^3	212	195	198	1.41	1.3	25×10^3	3540×10^3	3850×10^3

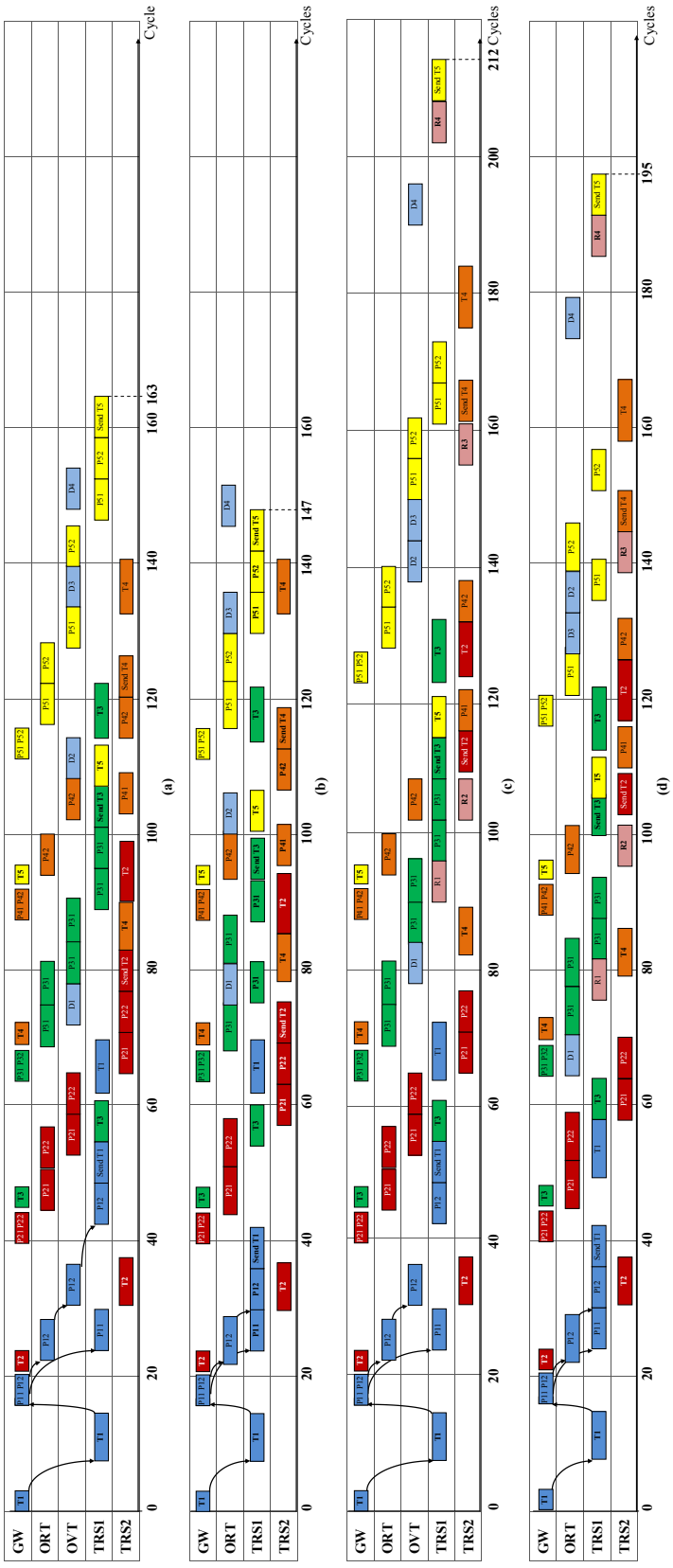

Fig. 5. Time scheduling of five tasks in the hardware pipeline of the Task Superscalar, a) five non-dependent tasks on base prototype, b) five non-dependent tasks on improved prototype, c) five dependent tasks on base prototype, b) five dependent tasks on improved prototype. (`Ti`: The required number of cycles that each component needs for processing packets related to Task i, `Pij`: The required number of cycles that each component needs for processing packets related to Parameter j of Task i, `Send Ti`: The required number of cycles for preparing a packet for Task i for sending to execution, `Di`: The required number of cycles for processing a DropParam packet (The index is the order that this packet is created in Figures 3,4), `R1`: The required number of cycles for processing a Ready packet (The index is the order that this packet is created in Figures 3,4).

Figures 5-a and 5-b show that processing for the five non-dependent tasks on base and improved prototypes respectively, and Figures 5-c and 5-d for the five dependent tasks on base and improved prototypes. The results show that due to the tiled structure of the Task Superscalar architecture, most of the operations of processing a new task and its parameters, and a finished task are simultaneously accomplished.

5. Conclusions

In this paper, we present and analyze the first hardware implementations of the full Task Superscalar architecture. The Task Superscalar is a task-based hybrid dataflow/von-Neumann architecture designed to support the StarSs programming model, adapting the out-of-order principle for parallel executing of tasks. In order to do so, two different complete hardware designs capable of keeping up to 1024 in-flight tasks, have been simulated. Our results show that both our prototypes are around 100x faster than the real software run-time implementation (i.e., Nanos) when executed at about 10x less frequency. Our results also demonstrate that the improved hardware prototype of the Task Superscalar utilizes less hardware resources and requires fewer cycles for task processing.

We expect to synthesize the hardware implementations of the full Task Superscalar architecture on an FPGA and test it with real workloads, in a near future.

Acknowledgements

This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by the Generalitat de Catalunya (contract 2009-SGR-980), and by the European FP7 project TERAFLUX id. 249013, http://www.teraflux.eu. We would also like to thank the Xilinx University Program for its hardware and software donations.

References

[1] P. Bellens, J. M. Perez, R. M. Badia, J. Labarta, CellSs: A programming model for the Cell BE architecture, in: Supercomputing, 2006.
[2] J. Perez, R. Badia, J. Labarta, A dependency-aware task-based programming environment for multi-core architectures, in: Intl. Conf. on Cluster Computing, 2008, pp. 142–151.
[3] P. Bellens, J. M. Perez, F. Cabarcas, A. Ramirez, R. M. Badia, J. Labarta, CellSs: Scheduling techniques to better exploit memory hierarchy, Sci. Program. 17 (1-2) (2009) 77–95.
[4] M. C. Rinard, M. S. Lam, The design, implementation, and evaluation of Jade, ACM Trans. Program. Lang. Syst. 20 (3) (1998) 483–545.
[5] J. C. Jenista, Y. h. Eom, B. C. Demsky, OoOJava: Software Out-of-Order execution, in: ACM Symp. on Principles and practice of parallel programming, 2011, pp. 57–68.
[6] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, M. Valero, Task Superscalar: An Out-of-Order task pipeline, in: Intl. Symp. on Microarchitecture, 2010, pp. 89–100.
[7] Y. Etsion, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, M. Valero, Task Superscalar: Using processors as functional units, in: Hot Topics in Parallelism, 2010.
[8] F. Yazdanpanah, D. Jimenez-Gonzalez, C. Alvarez-Martinez, Y. Etsion, R. M. Badia, FPGA-based prototype of the Task Superscalar architecture, in: 7th HiPEAC Workshop of Reconfigurable Computing, 2013.
[9] M. Gonzalez, J. Balart, A. Duran, X. Martorell, E. Ayguade, Nanos Mercurium: A research compiler for OpenMP, in: European Workshop on OpenMP, 2004.
[10] R. M. Badia, Top down programming methodology and tools with StarSs - enabling scalable programming paradigms: Extended abstract, in: Workshop on Scalable algorithms for large-scale systems, 2011, pp. 19–20.
[11] S. Kumar, C. J. Hughes, A. Nguyen, Carbon: Architectural support for fine-grained parallelism on chip multiprocessors, in: Intl. Symp. on Computer Architecture, 2007, pp. 162–173.
[12] J. Hoogerbrugge, A. Terechko, A multithreaded multicore system for embedded media processing, Trans. on High-performance Embedded Architectures and Compilers 3 (2).
[13] M. Själander, A. Terechko, M. Duranton, A look-ahead task management unit for embedded multi-core architectures, in: Conf. on Digital System Design, 2008, pp. 149–157.
[14] G. Al-Kadi, A. S. Terechko, A hardware task scheduler for embedded video processing, in: Intl. Conf. on High Performance & Embedded Architectures & Compilers, 2009, pp. 140–152.
[15] J. Noguera, R. M. Badia, Multitasking on reconfigurable architectures: Microarchitecture support and dynamic scheduling, ACM Trans. Embed. Comput. Syst. 3 (2) (2004) 385–406.
[16] J. Noguera, R. M. Badia, System-level power-performance trade-offs in task scheduling for dynamically reconfigurable architectures, in: Intl. Conf. on Compilers, architectures and synthesis for embedded systems, 2003, pp. 73–83.
[17] C. Meenderinck, B. Juurlink, A case for hardware task management support for the StarSs programming model, in: Conf. on Digital System Design, 2010, pp. 347–354.
[18] C. Meenderinck, B. Juurlink, Nexus: Hardware support for task-based programming, in: Conf. on Digital System Design, 2011, pp. 442–445.
[19] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, J. Labarta, Productive cluster programming with OmpSs, in: Euro-Par, 2011, pp. 555–566.