FPGA-Based Prototype of the Task Superscalar Architecture

Fahimeh Yazdanpanah[1,2], Daniel Jimenez-Gonzalez[1,2], Carlos Alvarez-Martinez[1,2], Yoav Etsion[3], Rosa M. Badia[1,2]

[1]Universitat Politècnica de Catalunya (UPC)
[2]Barcelona Supercomputing Center (BSC)
[3]Technion – Israel Institute of Technology
{fahimeh, djimenez, calvarez}@ac.upc.edu
yetsion@tce.technion.ac.il, rosa.m.badia@bsc.es

Abstract. In this paper, we present the first hardware implementation of a prototype of the Task Superscalar architecture; an experimental task-based dataflow scheduler that dynamically detects inter-task data dependencies, identifies task-level parallelism, and executes tasks out-of-order. The implemented hardware is based on a distributed design that can operate in parallel and is easily scalable to manage hundreds of cores in the same way that Out-of-Order architectures manage functional units. Our prototype operates at near 150Mhz, fits in a current commercial FPGA board, and can maintain up to 1024 in-flight tasks, managing the data dependencies in few cycles.

Keywords: Task Superscalar, Hardware task scheduler, FPGA, VHDL

1 Introduction

Exploiting concurrency to achieve greater performance is a difficult and important challenge for current high performance systems. Exploiting concurrency implies breaking a problem into tasks, and managing and coordinating them to ensure their correct execution simultaneously or interleaved in one or (probably) more processing units. Although the theory is plain, the complexity of traditional parallel programming models in most cases impedes the programmer to harvest performance.

To overcome this limitation, task-based dataflow programming models, such as StarSs [3, 4, 13], Jade [14], OoOJava [9], use dataflow principles to improve task-level parallelism (TLP). In these models, the programmer decomposes the program into short sequential code segments, referred to as *tasks*, which can potentially execute in parallel. Task-based programming models implicitly schedule work and data, thereby relieving the programmer of explicitly managing parallelism. In fact, they share conceptual similarities with out-of-order superscalar pipelines, such as dynamic data dependency analysis and dataflow scheduling, using tasks instead of instructions as a basic work unit. However, those programming models 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 creation and management of the execution of such tasks (henceforth called *task scheduling*) in software introduces overheads, and so becomes increasingly inefficient with the number of cores. In contrast, a hardware scheduling solution can achieve greater speed-ups because a hardware task scheduler requires few cycles for both scheduling and task synchronization. Furthermore, a tiled hardware task scheduler will be more scalable and more parallel than the equivalent software.

Task Superscalar [6,7] is a hybrid dataflow/von-Neumann design that implements the StarSs programming model. By employing task-based dataflow programming constructs, the Task Superscalar identifies inter-task dependencies, constructs the data dependency graph on runtime and dynamically schedules tasks for execution, thus, managing cores as functional units. The Task Superscalar, therefore, provides programmers with the same abstraction that makes Out-of-Order processors so appealing: a seemingly sequential interface for a parallel execution engine. The design of the Task Superscalar is a pipelined frontend that can be embedded into virtually any many-core fabric and manage it as a *Task Superscalar multiprocessor*.

The Task Superscalar had been implemented in software with all the limitations of a software implementation: limited parallelism and high memory consumption. A hardware implementation

will increase its speed and parallelism, reducing the power consumption at the same time. This may be used as a tightly couple accelerator or as an external task dispatcher to a many-core fabric. In this paper, we present the evolution of the original design of the Task Superscalar, and show the preliminary results for the first prototype of that design. To this end, the design has been adapted to accomplish the requisites of a real implementation, written in VHDL, simulated, and synthesized into a field programmable gate array (FPGA).

The reminder of the paper is organized as follows: Section 2 outlines state-of-the-art of hardware task schedulers. In Section 3, we overview the Task Superscalar architecture as the baseline of our design. Section 4 describes the design and operational flow of the pipeline of the hardware architecture. Then, in Section 5 we present the main issues arose by the hardware implementation of the Task Superscalar and the results obtained by the simulator and synthesis tools. Finally, conclusions are presented in Section 6.

2 Related Work

Static task management systems are not able to adapt to the variable behaviour of modern algorithms [1] due to the blocking nature of synchronization. To overcome this problem, different dynamic software task management systems (i.e., runtime systems) have been proposed like StarSs [3, 4, 13], OoOJava [9] or JADE [14]. These models try to support dynamic task creation and scheduling with a simple programming model [3]. However their flexibility comes at the cost of a rather laborious task management that should be done at runtime [2]. The cost of that potentially huge task management affects the scalability and performance of such systems, and potentially limits their applicability to applications with a large number of tasks.

Some hardware support solutions have been proposed to speed-up the task management, but most of them only schedule independent tasks, leaving it to the programmer to deliver tasks at the appropriate time. One example is Carbon [10], which minimizes task queuing overhead by implementing task queue operations and scheduling in hardware to support fast tasks dispatch and stealing. Indeed, TriMedia-based multicore system [8] contains a centralized task scheduling unit based on Carbon.

A look-ahead task management unit called TMU [15] has been proposed for reducing the task retrieval latency. TMU is a programmable task management unit that accelerates task creation and synchronization in hardware similar to video-oriented task schedulers [1]. Also, application specific instruction set processors (ASIP) have been proposed to speed-up software task management [5].

Etsion et al. [6, 7] have designed a hardware support for the StarSs programming model called Task Superscalar. The Task Superscalar 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. Nexus++ [11,12] 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 maintain the programmability, generality and easiness of use of the programming model.

3 Overview of Task Superscalar Architecture

In this section, we briefly describe the *Task Superscalar* [6, 7] architecture in order to highlight its strengths, weaknesses and potential bottlenecks. The Task Superscalar is a task-based dataflow architecture, that generalizes for tasks the operational flow of the instruction scheduler of Out-of-Order processors. A task may be any part of the code that has been identified as it using StartSs programming model. Therefore, the Task Superscalar combines the effectiveness of Out-of-Order processors in uncovering parallelism together with the task abstraction, and thereby provides a unified management layer for CMPs which effectively employs processors as functional units.

The Task Superscalar architecture uses StarSs programming model to uncover task level parallelism. This programming model enables programmers to explicitly expose task side-effects by annotating the operands of a task as input, output, or inout. With these annotations the Task Superscalar pipeline dynamically detects inter-task data dependencies, identifies task-level parallelism, and executes tasks in out-of-order manner. Therefore, this design enables programmers to exploit many-core systems effectively, while simultaneously simplifying their programming model.

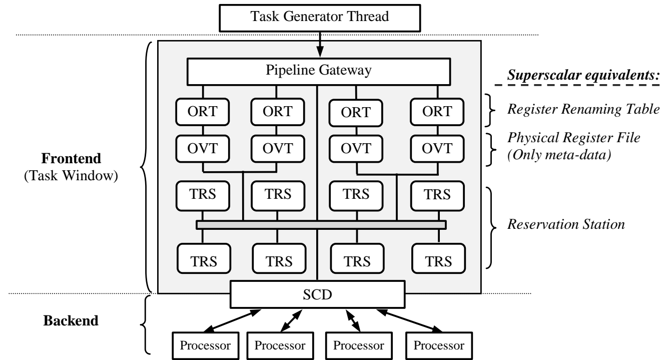

Fig. 1: The Task Superscalar architecture

The organization of the Task Superscalar architecture is illustrated in Figure 1. A task generator thread resolves the inter-task control path and sends tasks to the pipeline frontend for data dependency analysis. The pipeline frontend maintains a window of recently generated tasks that can contain tens of thousands of tasks. The frontend asynchronously decodes the task dependencies, generates the data dependency graph, and schedules tasks as they become ready (when all their parameters are available). Finally, ready tasks are sent to the backend for execution. The backend consists of a the task scheduler queuing system (SCD), and a many-core fabric.

As shown in Figure 1, the Task Superscalar frontend employs a tiled design, and is managed by an asynchronous point-to-point protocol. The frontend is composed of four different modules: Pipeline gateway (GW), task reservation stations (TRS), object renaming tables (ORT) and object versioning tables (OVT) that communicates between them using packets.

The GW is responsible for pushing the flow of tasks into the pipeline including: allocating TRS space for new tasks, distributing tasks 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, track the readiness of its parameters. To do this, 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 match memory parameters to the most recent task accessing them, and thereby detect object dependencies. Furthermore, each ORT has exactly one OVT associated with it. The OVT tracks all the live versions of every parameter that the associated ORT stores. That helps TRSs to maintain the data dependency graph. The functionality of the OVTs (and their associated ORT) resembles that of a physical register file, but only to maintain meta-data of parameters. Effectively, the OVT also manages anti- and output-dependencies, either through parameter renaming (changing register names to eliminate anti- and output dependencies), or by chaining different bidirectional (inout) parameters and unblocking them in-order. Versions are created for each input parameter that appears first time and for all output and inout parameters.

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 level of concurrency in the construction of the dataflow graph. This level of concurrency is a tradeoff between the basic latency associated when adding a new node to the graph, and the obtained overall throughput. Consequently, the rate in which nodes are added to the graph enables high task dispatch throughput. That is essential for utilizing large many-core fabrics.

Finally, the dispatch throughput requirements imposed on the Task Superscalar pipeline are further relaxed by the use of tasks as the basic execution unit. The longer execution time of tasks compared to that of instruction means that every dispatch operation occupies an execution unit for a few dozen microseconds, and thereby further amplifies the design's scalability.

4 Design and Operational Flow

The Task Superscalar architecture is composed by a set of independent modules that communicate with each other using messages (packets). In this section, we describe the operational flow of our

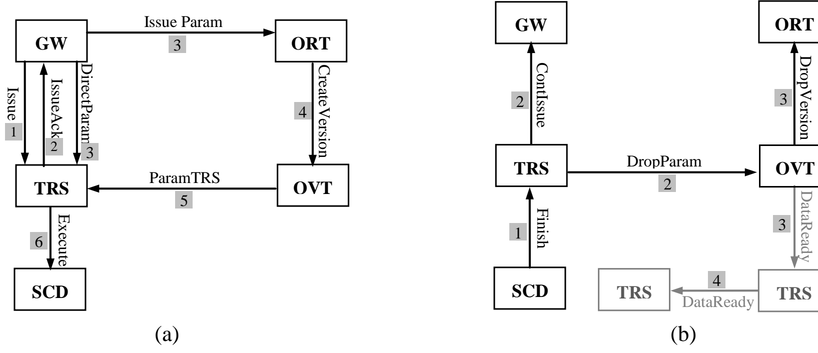

Fig. 2: Operational flow of Task Superscalar, a) when a task arrives to the pipeline, b) when a task finishes.

proposal for a real (hardware) implementation of the Task Superscalar architecture. Figure 2-a shows the general operational flow to add a new task to the scheduler while Figure 2-b shows the general process flow when a task is finished. We use those two figures to describe the general operational flow between the modules of the system. To help the explanation, the sequence order of the operations is annotated with labels on the arrows, close to the name of the communicated packet (message). Table 1 describes the packets that the modules use to communicate with each other.

In addition, we also present a more detailed description of the operational flow of each of the modules described in the general view. Figures 3, 4, 5, and 6 present the detail for the operational flow of the GW, TRS, ORT and OVT modules, respectively. All the modules (but GW), need to wait for a packet in their input FIFO before they can proceed to the following operations. GW needs to wait for new tasks. We will refer to those figures along the description of the general operational flow.

4.1 Task arrival operational flow

As Figure 2-a shows, the processing of a task begins when the GW sends an allocation request to one of the TRSs (`Issue` packet, sequence order 1 in Figure 2-a, i.e., Figure 2-a(1)). The operational flow of the GW module, shown in Figure 3, begins when the GW gets a task from the task generator thread and selects a TRS that has space for saving the task. If all the TRSs are full, the GW waits until a `ContIssue` packet arrives (Figure 2-b(2)), which means that a task has finished and there is free space in a TRS. After selecting a TRS, the GW sends to the TRS an `Issue` packet to allocate a task storage space for task meta-data. The operational flow of the TRS is shown in Figure 4. When a TRS gets an `Issue` packet, it allocates a slot for the task and its parameters, and sends an `IssueAck` packet (Figure 2-a(2)) to the GW for notifying the address of the allocated space.

Table 1: Definition of the packets.

Packet	Description	Source/Destination
ContIssue	notifies the GW that an space in the TRS memory is available.	TRS/GW
CreateVersion	is for creating and/or updating an OVT entry.	ORT/OVT
DataReady	notifies another task(s) that a parameter is ready.	TRS or OVT/TRS
DirectParam	includes a direct (scalar) parameter.	GW/TRS
DropParam	is sent for informing the releasing of the parameter.	TRS/OVT
DropVersion	is for getting the permission of releasing a version.	OVT/ORT
Execute	includes the meta-data of a ready task for executing.	TRS/SCD
Finish	notifies TRSs that execution of the task has been finished.	SCD/TRS
Issue	includes meta-data of a task.	GW/TRS
IssueAck	is an acknowledgement for task allocation.	TRS/GW
ParamORT	includes a non-scalar parameter for data dependency analysis.	GW/ORT
ParamTRS	is for sending a decoded parameter.	OVT/TRS

Once a TRS slot is allocated, the GW starts to send the parameters of the allocated task to the TRS. Scalar parameters are sent directly to the allocated TRS using `DirectParam` packet. For non-scalar parameters, the GW initiates the dependency decoding by distributing parameters to the ORTs. To do this, the GW sends for each non-scalar parameter a `ParamORT` packet (Figure 2-a(3)). If ORT (or OVT) does not have free space, GW waits until they have space as described in Figure 3.

The functionality of the ORT is similar to the register renaming table as Figure 5 shows. When it receives a `ParamORT` packet from the GW, the ORT checks if an entry for this parameter already exits,

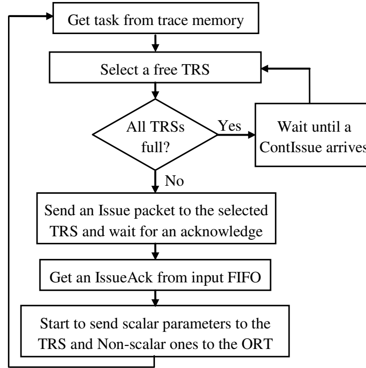

Fig. 3: Operational flow of GW

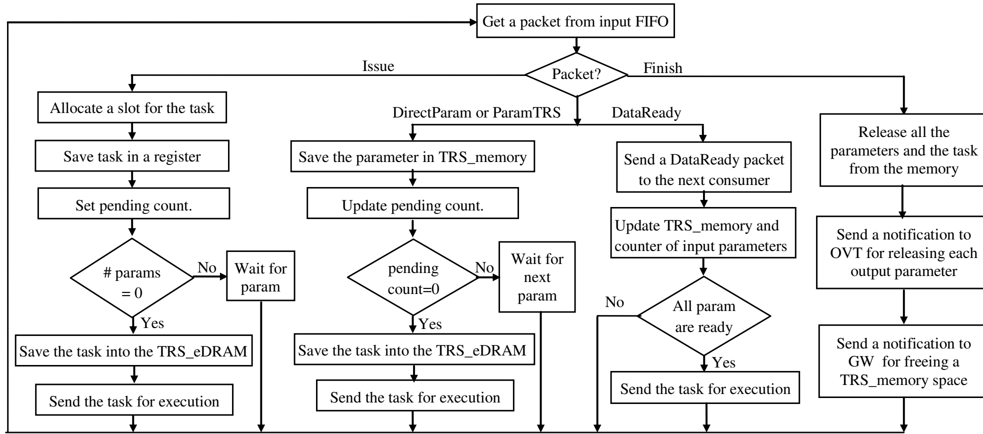

Fig. 4: The operational flow of TRS

and sends a `CreateVersion` packet to activate the OVT. If there was an entry for the parameter, the ORT updates it, otherwise, a new entry is created, as described in Figure 5.

When an OVT gets a `CreateVersion` packet from its ORT, as shown in Figure 6, it checks if it is an input or output parameter: for each output parameter, OVT creates a new version for it (a new producer) and updates the previous version of the parameter if it exists. For each input parameter, if it is the first time that the parameter appears, the OVT creates a new entry for it. Otherwise, the OVT only updates the version of the parameter adding a new consumer. Meanwhile, the OVT sends a `ParamTRS` packet to the TRS, which includes the information of the parameter, its previous user and its related version (Figure 2-a(5)).

When a TRS gets a `ParamTRS` packet, it updates the corresponding task information. When all the parameters of a task are ready, TRS send the task and its parameters to the SCD by an `Execute` packet (Figure 2-a(6)).

Since we do not use renaming method in this design, in any case (consumer or producer), we do not need to create a copy of the parameter an its original address is used. It is essential to note that without renaming, bidirectional (i.e., inout) parameters are processed similar to output parameters.

4.2 Task ending operational flow

Figure 2-b shows the procedure of finishing a task. When a task finishes, the corresponding TRS gets a `Finish` packet from the SCD. Then it starts to release the parameters of the task. That is done by sending a `DropParam` packet, for each of the parameters, to the OVT. After all the parameters of the task are released, the TRS frees the TRS_memory of the task and sends a `ContIssue` packet to the GW indicating that an slot has became free and there is space for allocating new tasks (Figure 2.b(2)).

At the same time, when the `DropParam` packet arrives to the OVT, it sends a `DropVersion` packet to ORT (Figure 2.b(3)). Also, for each output parameter, if there are consumers waiting, the OVT notifies the TRS that is on the top of the consumer stack that the parameter is ready by

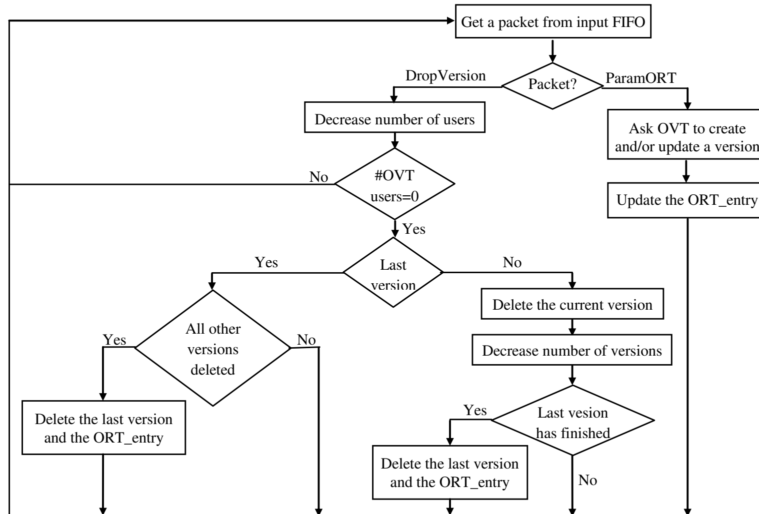

Fig. 5: The operational flow of ORT

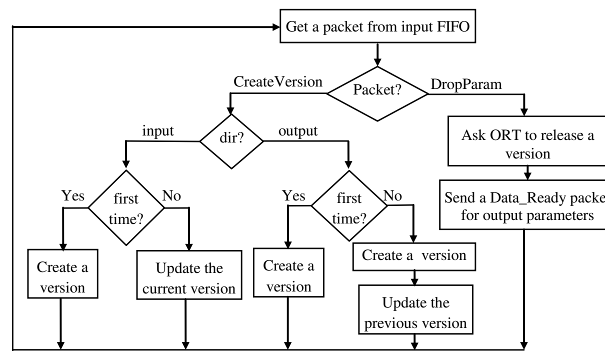

Fig. 6: The operational flow of OVT

sending a `DataReady` packet (Figure 2-b(3)). When a consumer TRS gets a `DataReady` packet for a parameter, it updates the associated memory entry and, if there is another TRS consumer, sends the `DataReady` packet to that TRS (Figure 2.b(4)). `DataReady` packet is propagated between TRS producer and TRS consumers based on consumer chaining that is repeated until there are no more consumers in the stack. When each of the consumers finishes, it sends also a `DropParam` packet to the OVT (Figure 2.b(2)). The OVT collects them and, after getting all of them (i.e., the version of the parameter is no longer used), it sends a `DataReady` packet to the next producer of the parameter if it exits.

Meanwhile, when the ORT receives a `DropVersion` packet (Figure 2.b(3)), it decreases the total number of users of the parameter. If there is no more users for the version it may be deleted (details of the ORT operational flow are shown in Figure 5). In the case that the version is the last and there is no more user for the parameter, the ORT entry is deleted and the version in the OVT is freed. In the case that the version is not the last one and there is no more users for the parameter, the ORT frees two entries of the OVT list (one for the version that is not going to be used and another for the last version one) and also deletes the corresponding ORT entry. This behaviour is due to the fact that the last version of a parameter should not be deleted until all previous (other) versions are deleted. The last version should be remained because if a new consumer arrives, it should be linked to the last version and not to the one in use. Therefore, as Figure 5 shows, in response to a `DropVersion` Packet, in some cases ORT frees one entry of the list, in some cases, it frees two entries, and in some cases it does not free any entry of the list.

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

5 Task Superscalar Prototype

We have implemented a hardware prototype of the Task Superscalar architecture based on the operational flow that has been described in Section 4. The implemented prototype is an adaption of the high level description of the pipeline of the Task Superscalar into VHDL code. Figure 7 illustrates the high-level diagram of the implemented version of the design. The prototype is mainly composed of one GW, two TRSs, one ORT, one OVT and one SCD.

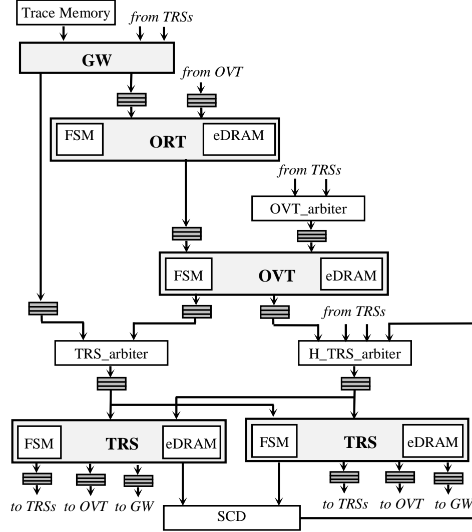

Fig. 7: Task Superscalar prototype

One main consideration 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. Those memory units, eDRAM memories in the original Task Superscalar design [6, 7], have been mapped into distributed RAM of the FPGA. Another important consideration is minimizing the cycles required for processing input packets in order to increase the overall system speed.

The main functionality of each component is done by a finite state machine (FSM). The FSMs are designed in such a manner that each state is done in one clock cycle. The TRS FSM processes different packets from other components of the pipeline. Those packets include the information of tasks (`Issue` packet), their parameters (`DirectParam` and `ParamTRS` packets), notification of termination of execution a task (`Finish` packet) and readiness of output parameters (`DataReady` packet). Since processing of `Finish` and `DataReady` packets have higher priority than other packets, TRSs get these packets from a separate FIFO which has higher priority than the other input FIFO of TRSs. The ORT FSM controls the creation and deletion of versions of parameters stored in its corresponding OVT. The ORT has two input FIFOs: one high priority FIFO for `DropVersion` packets and the other, with normal priority, for `ParamORT` packets. The OVT FSM is responsible for creating and updating the parameters based on the decisions of the ORT. Then, the OVT sends the parameters information to the associated TRS. OVT has also two input FIFOs: one high priority FIFO for `DropParam` packets and the other, with normal priority, for `CreateVersion` packets.

For the prototype, a trace memory, implemented with RAM, is used instead of a task generator thread in order to test and simulate the functionality of the pipeline. The trace memory can save up to 1024 tasks. Each task includes 17 80-bit entries, two for meta-data of a task and the others for up to 15 parameters.

Our prototype is an FPGA memory conscious design. As mentioned, memory is one of the critical FPGA resources. The memory of the OVT can save up to 8k versions of parameters. The ORT memory has 1024 sets and each set has 8 ways. It can store the meta-data of 8K parameters. TRS memory is divided into blocks (slots). Each block has 16 200-bit entries, one for every meta-data of the task. With this size, it 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). The whole prototype with two TRSs can store up to 1024 in-flight tasks.

The interconnection network is also an important component since it can easily limit the scalability of the design. To overcome that potential limitation problem, the network includes arbiters and 4-element FIFOs that decouple the processing of every component in the system. Those FIFOs are divided, mainly, in high and low priority FIFOs. The packets that free resources on the system (e.g., `Finish` or `DropParam` packets) go to the high priority FIFOs while the packets that allocate resources go to the low priority FIFOs. That organization allows the system to scale avoiding the possibility of deadlocks. In order to minimize the width of FIFOs and also the bandwidth of intra-module buses, we define packets with the minimum possible size multiple of 8 bits (Table 3). The width of the memory data, the size of the registers and all the packet sizes are multiple of eight in order to obtain the best place and route result in the target FPGA. In the prototype, each message includes a packet. The only corner case that makes an exception to this rule is the `Execute` packet. The size of `Execute` packet is related to the number of parameters and the maximum size of this packet could be up to 1295 bits. In this case, in the prototype design we divided the packet into messages of up to 328 bits due to implementation constraints. Hence, a ready task and its parameters are sent to the SCD in several cycles based on the number of parameters of the task.

5.1 Experimental Setup

To verify the functionality of each module, we have simulated the hardware design in ModelSim 6.6d. We have simulated them using several bit traces. Those traces represent input packets that help to test the main and corner (e.g., chains of inout parameters) cases. We have tested the correctness of the output packets generated by the modules and also the modifications to their related memories.

After validating their correct behavior, the modules of the hardware design have been synthesized using the Xilinx synthesis technology tool(XST) version 14.2. For synthesizing purpose, we have selected two devices from the Xilinx Virtex 7 family: first the xc7vh290t that has the smallest number of LUTs among the available devices of this family; the second selected device is the xc7v2000t that is the biggest one in the family. Table 2 presents some information of these devices including the number of slice flip flops (FFs) and slice look-up tables (LUTs).

Table 2: Device Information

Device	Package	# Slice look-up tables (LUTs)	# Slice flip flops (FFs)	# Logic cells	Max distributed RAM (Kb)	Max block RAM (Kb)
xc7vh290t	HCG1155	218800	437600	284000	4425	16920
xc7v2000t	FLG1925	1221600	2443200	1954560	21550	46512

5.2 Simulation and Synthesis Results

Here, we present and analyze the results obtained by the simulation and synthesis of the hardware prototype of the Task Superscalar architecture.

Table 3 presents the required latency (in cycles) for each FSMs to process the received packets. The computed latencies take into account that most of the operations of the FSMs are overlapped with each other (i.e., performed in parallel). We have tested each module with its different kinds of input packets. The presented results are obtained using waveform outputs. The processing of each packet, based on the definition of the packet, may have an access to the memory and produce another packet, if required. Therefore, the processing of a packet may contribute with different latencies depending on the tasks and conditions of the system. In addition to the aforementioned latencies, each module uses four cycles in order to read the input packets, select the one with the highest priority and initialize the FSM.

Table 4 presents the synthesis results and characteristics of the memories of the synthesized hardware. The modules are completely synchronous. The frequency shown in the table (Freq. column) is the maximum frequency of the clock signal that each module is synchronized with. Slice logic utilization includes the number of slice flip flops (FFs, the number of one-bit registers used in the whole FPGA) and slice look-up tables (LUTs) that are used as logic or distributed RAM, or even as shift registers. Macro statistics column includes the number of required resources such as registers (R), comparators (C), multiplexers (M), adders/subtractors (A), tri-state buffers (T) and xor gates (X). Finally, the table presents the percentage of usage that every RAM module represents in each target FPGA, and some configuration details about the implemented memory such as the memory capacity, associativity, entry size and number of entries.

The design of the RAM modules has been done in such a way that, accessing to the memory components takes two pipelined cycles: one cycle for setting the control signals (enabling access

Table 3: Latencies of processing the packets

Packet	Processing latency	Responsible Unit	Size (bits)
ContIssue	2 cycles	GW	24
CreateVersion	- 2 cycles if the (input or output) parameter appears for the first time - 3 cycles if the input parameter does not appear for the first time - 4 cycles if the output parameter does not appear for the first time	OVT	240
DataReady	2 cycles	TRS	64
DropParam	2 cycles	OVT	40
DropVersion	- 2 cycles for deleting the last version - 3 cycles for deleting the last version and the previous one	ORT	120
Finish	3 cycles + 2 additional cycles for loading every parameter	TRS	88
Forming Execute packet	1 cycle (for loading meta-data of a task) + 2 cycles for loading every parameter + 1 cycle for sending the task to the SCD	TRS	170+(#param*75)
IssueAck	2 cycles	GW	24
Issue	- 4 cycles for tasks without any parameter - 3 cycles for tasks with at least parameter + 2 additional cycles for every parameter (using ParamTRS (200bits) or DirectParam (96 bits) packets)	TRS	160
ParamORT	2 cycles	ORT	168

mode) and the other for accomplishing the operation (e.g., writing or reading). To have a performant FPGA design, the memory components have synchronous read and write enables [1]. In all of the cases, the cycle of setting the control signals has been overlapped with the other previous operations of the FSMs.

Table 4: Statictics of the eDRAMs (R:registers, M:multiplexer, C:comparator, T:tristate buffers, X:xor)

Component	Freq. (MHz)	Slice logic utilization		Macro Statistics	Usage of xc7vh290t	Usage of xc7v2000t	Memory characteristics			
		#Slice LUTs	#Slice FFs				Capacity (KBytes)	Associativity	Entry size (bits)	#Entries
TRS_eDRAM	203.18	34358	710	499R,511M, 497X	16%	2.8%	200	Direct mapped	200	8K
OVT_eDRAM	348.03	22112	160	1R	10%	1,8%	160	Direct mapped	160	8K
ORT_eDRAM	262.42	24750	9607	7238R,17C,107 6M256T,24X	11.3%	2%	200	8-way set associative	146 (+ 54 for tag)	8K

Table 5 presents the synthesis results of the main modules (with their memory included) and also the resource utilization of all FIFOs and arbiters in the prototype. We have also made an estimation of how many resources will be used by the whole system once all the modules are connected. One of the main considerations in designing this prototype was its scalability. As shown in table 5 the resource usage shows that for implementing a system similar to the Figure 7, the design will take about 53% of the smallest device (i.e., xc7vh290t). On the other hand, it can be replicated up to ten times (probably a little less due to network constraints) in the largest one (i.e., xc7v2000t). These results allow us to be optimistic about the possibilities of design trade-off evaluation of the whole system (i.e., optimal number of TRSs vs. number of pairs OVT-ORT, memory limits, etc.).

Table 5: Synthesis result of the main modules (R:registers, M:multiplexer, A:adder/subtractor, C:comparator, T:tristate buffers, X:xor)

Component	Freq. (MHz)	Slice logic utilization		Macro Statistics	Usage of xc7vh290t	Usage of xc7v2000t
		#Slice LUTs	#Slice FFs			
GW	389.63	4756	600	455R,4A,198M	2%	0.3%
TRS	146.93	39771	4800	1970R, 9A, 2C, 945M, 497X	18.1%	3.2%
OVT	297.40	26843	2917	919R, 1A, 266M	12.3%	2.2%
ORT	201.57	28067	13654	8856R, 21C, 1137M, 8427X	13.0%	2.3%
FIFOs	498.32	8190	4770	108R, 54A, 90M	3.7%	0.6%
Arbiters	703.60	3102	2397	786R, 771M, 768T	1.4%	0.25%
Total	146.93	110729	29138	13094R, 3407M, 68A, 268T, 8924X	51.4%	9%

6 Conclusions

In this paper, we present the first hardware implementation of the Task Superscalar; a task-based dataflow architecture designed to support the StarSs programming model. This architecture extends

[1] According to XST user guide [16], "a VHDL RAM with synchronous write and read can be synthesized in both BRAM and distributed styles, while with asynchronous read must be in distributed style."

the out-of-order notion of executing in parallel sequentially written instructions to tasks. To do so, the Task Superscalar architecture creates a large execution window of up to tens of thousands of tasks that are scheduled to execute as soon as their parameters become available.

In order to be able to scale to this extent, the Task Superscalar architecture is designed as a set of different independent processing modules that keeps track of the information on the system (tasks, parameters, versions, etc.) and communicate asynchronously with each other. The resulting design can be embedded into any manycore fabric and manage its processors in the same way that an Out-of-Order processor manages arithmetic units.

To implement the Task Superscalar in hardware, its design has been modified to fit the limitations of an FPGA based implementation. In this paper, we present the basis of the Task Superscalar architecture and the complete modified operational flow of the implemented system while discussing the design decisions. All the modules that compose the system have been tested, simulated to verify their behaviour and synthesized to two Xilinx FPGAs. Our results show that the prototype design (that can manage up to 1024 in-flight tasks) can fit in any current commercial FPGA. Moreover, with a high-end fabric it can be replicated several times in order to increase the task window to a length enough to fulfill the needs of a large supercomputer. All the modules at the proposed implementation can run at nearly 150 MHz and are able to process packets in, at most, 5 cycles (plus 2 extra cycles per task parameter).

As a future work, we will join all the components of the prototype and simulate and synthesis the whole design and evaluate the hardware prototype using real workloads.

7 Acknowledgemnt

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. G. Al-Kadi and A. S. Terechko, "A hardware task scheduler for embedded video processing," in *Intl. Conf. on High Performance & Embedded Architectures & Compilers*, 2009, pp. 140–152.
2. 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.
3. P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a programming model for the Cell BE architecture," in *Supercomputing*, 2006.
4. P. Bellens, J. M. Perez, F. Cabarcas, A. Ramirez, R. M. Badia, and J. Labarta, "CellSs: Scheduling techniques to better exploit memory hierarchy."
5. J. Castrillon, D. Zhang, T. Kempf, B. Vanthournout, R. Leupers, and G. Ascheid, "Task management in MPSoCs: an ASIP approach," in *Intl. Conf. on Computer-Aided Design*, 2009, pp. 587–594.
6. 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 *Intl. Symp. on Microarchitecture*, 2010, pp. 89–100.
7. Y. Etsion, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Task Superscalar: Using processors as functional units," in *Hot Topics in Parallelism*, 2010.
8. J. Hoogerbrugge and A. Terechko, "A multithreaded multicore system for embedded media processing," *Trans. on High-performance Embedded Architectures and Compilers*, vol. 3, no. 2, 2011.
9. J. C. Jenista, Y. h. Eom, and B. C. Demsky, "OoOJava: software Out-of-Order execution," in *ACM Symp. on Principles and practice of parallel programming*, 2011, pp. 57–68.
10. S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *Intl. Symp. on Computer Architecture*, 2007, pp. 162–173.
11. C. Meenderinck and B. Juurlink, "A case for hardware task management support for the StarSs programming model," in *Conf. on Digital System Design*, 2010, pp. 347–354.
12. C. Meenderinck and B. Juurlink, "Nexus: Hardware support for task-based programming," in *Conf. on Digital System Design*, 2011, pp. 442–445.
13. J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Intl. Conf. on Cluster Computing*, 2008, pp. 142–151.
14. M. C. Rinard and M. S. Lam, "The design, implementation, and evaluation of Jade," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 3, pp. 483–545, 1998.
15. M. Själander, A. Terechko, and M. Duranton, "A look-ahead task management unit for embedded multicore architectures," in *Conf. on Digital System Design*, 2008, pp. 149–157.
16. www.xilinx.com, "XST user guide for Virtex-6, Spartan-6, and 7 series devices (v 14.1)," 2012.