Hardware Design of Task Superscalar Architecture

Ph. D. Candidate: **Fahimeh Yazdanpanah**

Supervisors: Dr. Carlos Alvarez Martinez and Dr. Daniel Jimenez Gonzalez

Computer Architecture Department, Universitat Politecnica de Catalunya

A thesis submitted for the degree of

Doctoral (PhD)

2014

Abstract

Exploiting concurrency 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 at task granularity. In this sense, different dynamic software task management systems, such as task-based dataflow programming models, 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 when there is fine-grained task granularity and a large amount of tasks.

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 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.

The Task Superscalar is a hybrid dataflow/von-Neumann architecture that exploits task level parallelism of the program. The Task Superscalar com-

bines the effectiveness of Out-of-Order processors together with the task abstraction, and thereby provides an 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.

In this thesis, a Hardware Task Superscalar architecture is designed to be integrated in a future High Performance Computer with the ability to exploit fine-grained task parallelism. The main contributions of this thesis are: (1) a design of the operational flow of Task Superscalar architecture adapted and improved for hardware implementation, (2) a HDL prototype for latency exploration, (3) a full cycle-accurate simulator of the Hardware Task Superscalar (based on the previously obtained latencies), (4) full design space exploration of the Task Superscalar component configuration (number and size) for systems with different number of processing elements (cores), (5) comparison with a software implementation of a real task-based programming model runtime using real benchmarks, and (6) hardware resource usage exploration of the selected configurations.

Keywords: Hardware Task Superscalar (HTSS), SimTSS, OmpSs, Nanos++ Runtime System, Hybrid Dataflow/von-Neumann, Task Scheduler

Acknowledgements

This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2007-60625 and TIN2012-34557, 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.

Contents

List of Figures

LIST OF FIGURES

List of Tables

Glossary and Abbreviation

Glossary

In this glossary section, the essential terms which are used in this thesis are presented. The glossary has been included as having many definitions in the main body of the thesis can be disruptive and hinder the smooth flow of ideas for the reader.

- **Big Configuration (BigConf).** BigConf is an HTSS configuration with more-than-enough resources that provides high performance for large many-core systems.
- **BRAM memory style.** BRAM memory style is a method for synthesizing a memory module onto blocks of RAMs of an FPGA.
- **Control Flow/Dataflow Class.** Models in Control Flow/Dataflow Class schedule the instructions within a block in Dataflow manner, whereas blocks are scheduled in control flow manner.
- **Dataflow graph (DFG).** DFG consists of named nodes and arcs that represent instructions and data dependencies among instructions.
- **Dataflow/Control Flow Class.** Models in Dataflow/Control Flow class employ dataflow rules between blocks of instructions and control flow scheduling inside the blocks of instructions.
- **Distributed memory style.** Distributed memory style is one manner for synthesizing memory modules. Using this style, the memory module is synthesized and mapped onto the lookup tables (LUTs) for memory units of an FPGA.

- **Dependence Memory (DM).** DM keeps the dependence information of incoming tasks to the pipeline of the Task Superscalar architecture.
- **Enhanced Control Flow Class.** Models in Enhanced Control Flow Class schedule blocks in control flow manner, whereas the instructions within a block are scheduled in a mixed approach of control flow and Dataflow manner.
- **Enhanced Dataflow Class.** Models in Enhanced Dataflow Class use Dataflow firing rules for instructions inside the blocks and for the blocks themselves.
- **extended ORT (eORT).** eORT is responsible for storing dependencies and their versions in order to manage data dependency analysis.
- **Gateway (GW).** GW is responsible for issuing the tasks and their dependences to the pipeline.
- **High Performance Computing Configuration (HPCConf).** HPCConf is an HTSS configuration suitable for high performance computing, many-core systems with the least resources that provides high performance.
- **Hardware Task SuperScalar (HTSS).** HTSS is a modified version of the Task Superscalar architecture for hardware implementation.
- **Hybrid dataflow/von-Neumann models.** Hybrid dataflow/von-Neumann models try to harness the parallelism and data synchronization inherent to dataflow models, while maintaining existing programming methodology and abstractions that are largely based on von-Neumann models.
- **improved Gateway (iGW).** iGW is an improved version of the GW with less latency than GW.
- **improved TRS (iTRS).** iTRS is an improved version of TRS for HTSS.3.
- **Mapping.** The goal of the mapping process is to associate efficiently the functionality of the application to the target platform. It is a combined task consisting of allocation and scheduling of the operations to operators.

- **Minimum Configuration (MinConf).** MinConf is an HTSS configuration suitable for small multi-core systems with one GW, one eORT, one TRS and one TS.
- **Object Reservation Table (ORT).** ORT manages data dependency analysis by saving meta-data of dependences.
- **Object Version Table (OVT).** OVT is responsible for managing the versions of the dependences.
- **Processing Element (PE).** At any abstraction levels, a processing element refers to a unit which is able to process data. E.g. it can be a computer in a network, a FPGA or DSP in a system, or at a lower level an operator inside a DSP or FPGA.
- **Scheduling.** Consists in deciding in which order the operations of an algorithm should execute.
- **SimTSS.** SimTSS is a full cycle-accurate simulator of the Hardware Task Superscalar.
- **Synthesis.** The synthesis task is responsible for transforming a behavioural description (e.g. C or Behavioural VHDL) into a dedicated hardware block. Although many articles as well as EDA tools consider the synthesis to include only the creation and instantiation of operators in logic gates, here we also include the place and route task.
- **Task Parallelism.** Task parallelism (or function parallelism) emphasizes on distributing execution processes across different parallel computing nodes. In the task parallelism, the program is partitioned into cooperative tasks. Tasks can accept inputs as a prerequisite to their start, and when they terminate send results to other tasks. Each task can execute a different set of functions and all tasks can run asynchronously. Tasks can generate other tasks dynamically based on data dependency analysis. Such collections of tasks may be represented by a direct acyclic graph (DAG), in which nodes represent tasks and arcs represent communication (data dependencies).

- **Task Scheduling.** Task scheduling refers to the way tasks are assigned to run on the available processing elements. It can be statically at compile-time or dynamically at runtime. A static scheduler collects some statically known data, such as task arrival time and task execution time, and uses these data to decide the task execution sequence. The dynamic scheduler is able to schedule dynamically arrived tasks. However, it leads to some runtime computation overhead.
- T_∞**.** T_∞ simulation measures the maximum performance that can be obtained by a task parallel strategy when infinite resources are available.
- **Task memory (TM).** TM is embedded in a TRS for storing meta-data of in-flight tasks and their dependences.
- **Task Reservation Station (TRS).** TRS is responsible for managing in-flight tasks.
- **Task Scheduler (TS).** TS is responsible for distributing ready tasks to the worker processors.
- **Version Memory (VM).** VM stores the versions of the dependences.
- **ZeroHTSS.** ZeroHTSS is an HTSS with unlimited number of resources, where packets are processed in each FSM at cost zero.

List of Abbreviations

BB	Basic Block of Instructions
BigConf	Big Configuration
BRAM	Block RAM
BSC	Barcelona Supercomputing Center
BSize	Block Size
C-Cores	Conservation Cores
CF	Control Flow
CLB	Configurable Logic Block
CMP	Chip-level MultiProcessor
CurConf	Current HTSS Configuration
DAG	Direct Acyclic Graph
DDM	Data Driven Machine
DF	DataFlow
DFG	DataFlow Graph
DLP	Data Level Parallelism
DM	Dependence Memory
DySER	Dynamically Specialized Execution Resource
eORT	extended Object Renaming Tables
ETS	Explicit Token Store

GLOSSARY

FF	Flip Flop
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPU	Graphics Processing Unit
GW	Gateway
HDL	Hardware Description Language
HPC	High Performance Computing
HPCConf	High Performance Computing Configuration
HPL	High Performance Linpack
HTSS	Hardware Task SuperScalar
iGW	improved Gateway
ILP	Instruction Level Parallelism
IOB	Input Output Buffer
iTRS	improved
LUT	Look-Up Tables
MinConf	Minimum Configurtion
NB	Number of Block
OoO	Out-of-Order
ORT	Object Renaming Table
OVT	Object Versioning Table
PC	Program Counter
PE	Processing Element
RTL	Register Transfer Level

SDF	Scheduled Dataflow
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SimTSS	Simulator of Task Superscalar
SMP	Symmetric Multi-Processor
SMT	Simultaneous Multi-Threading
SoC	System-on-Chip
TaskID	Task Identifier
TLP	Thread Level Parallelism
TLS	Thread Level Speculation
TM	Task Memory
TMU	Task Management Unit
TPC	Tagged Procedure Calls
TRS	Task Reservation Station
TS	Task Scheduler
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VM	Version Memory
XST	Xilinx Synthesis Technology

Chapter 1

Introduction

1.1 Motivations

Current computing systems face the end of Dennard scaling while keep increasing the number of transistors (Moore' law). This leads to chips that are hitting a power wall because of slowed supply voltage scaling. One of the approaches for tackling this challenge is the use of homogeneous and heterogeneous multi-core architectures. These multi-core architectures are conventionally based on the von-Neumann (traditional control flow) computing model, which is inherently sequential because of its use of a program counter and an updateable memory. Nevertheless, the von-Neumann computing model is able to exploit some limited parallelism in different levels: instruction level parallelism (ILP), data level parallelism (DLP), and thread level parallelism (TLP). While instruction level (fine-grained) parallelism is usually discovered by the hardware, DLP is more dependent on the programmer or the compiler and TLP is usually totally dependent on the programmer (with same hardware support).

Dataflow computing model represents a radical alternative to the von-Neumann computing model, offering many opportunities for parallel processing. But this model has not become mainstream in the world of general purpose processors and programming because of its inability to efficiently support data structures and imperative programming languages. However, it has seriously influenced parallel computing, and its techniques have found their way into many products such as Out-of-Order processors that mix superscalar approach with dataflow concept. Moreover, in order to increase their performance and power efficiency, systems can be designed as hybrid architectures

that combine the dataflow and von-Neumann models of computation.

Regardless of the chosen model, exploiting concurrency consists in breaking a problem into discrete parts (that can be called *tasks* when they are composed of several instructions), and managing and coordinating them to ensure correct execution, simultaneously or interleaved in one or more processing units. Although the simple definition, exploiting concurrency is a difficult and important challenge for current high performance systems. In this context, several software schedulers have been proposed to exploit concurrency efficiently. Recently, there has been a growing interest in developing runtime task scheduling techniques due to their flexibility and high performance capability. 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.

general-purpose dataflow task-based programming model that simplifies parallel programmers' life is OmpSs [1] It benefits dynamic data dependency analysis, dataflow scheduling and out-of-order executing. OmpSs has been implemented in software through Mercurium compiler and Nanos++ runtime system. Although the software implementation is optimized, it introduces some more overhead in task execution. This limits the efficiency for small tasks, a problem that increases with the number of available cores as it is shown in Figures 1.1 and 1.2. Figure 1.1 shows the average task size and the number of tasks of different OmpSs applications (Cholesky, SparseLU, Heat and LU) for a problem size of 2048×2048 elements when varying the granularity (block size) of tasks. Figure 1.2 shows the speed-up obtained for these same problems in a real execution with 12 cores. As it can be seen, when the number of small tasks significantly increases in Figure 1.1, the overall speed-ups of the OmpSs applications dramatically decrease in Figure 1.2. In contrast, a tiled hardware task scheduler would be more efficient for small tasks and would provide larger task throughput.

The Task Superscalar [6, 7] is a hybrid dataflow/von-Neumann architecture [8] that supports the OmpSs programming model as a hardware task scheduler. This architec-

[1] OpenMP 4.0 is also a dataflow task-based programming model that has just appeared (May 2014). This model has been very influenced by the OmpSs programming model.

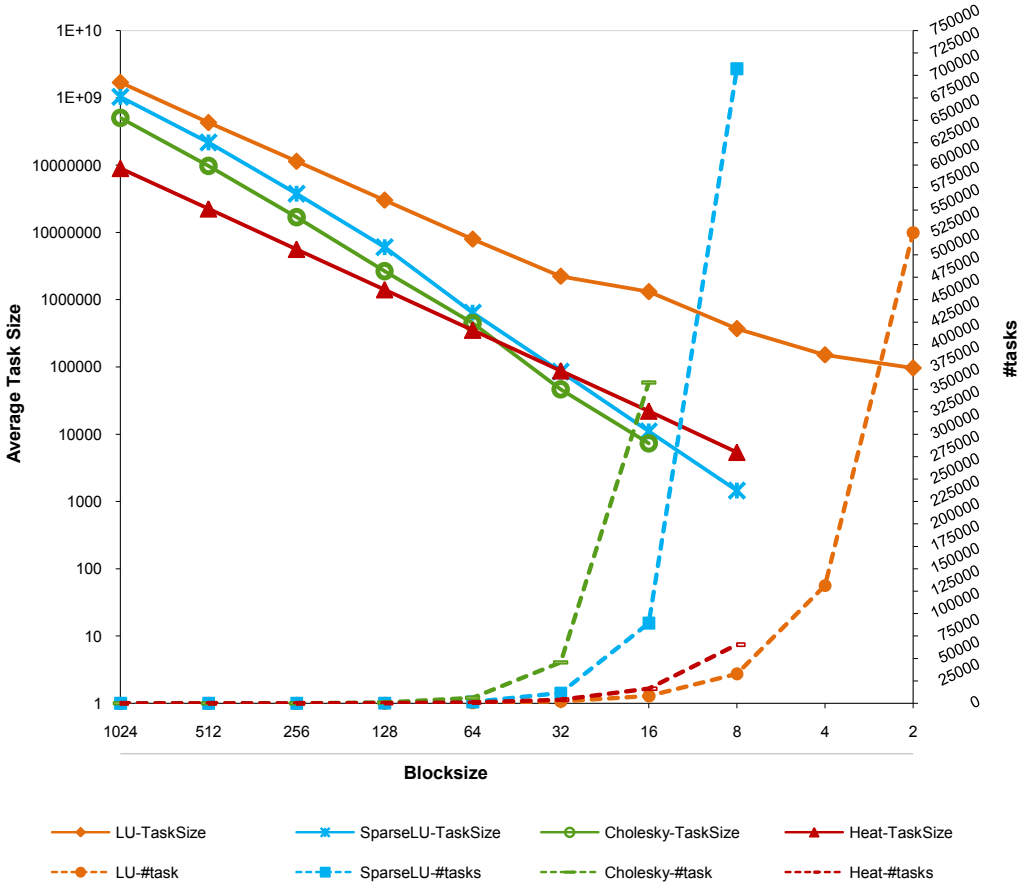


Figure 1.1: Average task size and the number of tasks of different OmpSs applications for 2048×2048 elements with different block size of tasks

ture takes benefits of the effectiveness of Out-of-Order processors applied to the task level parallelism of the program.

1.1.1 Problem Statement

The initial design of the Task Superscalar architecture had only been simulated in software with limited parallelism and high memory consumption due to the nature of the software implementation. Although that approach demonstrated the validity of the idea, a real working proof of concept was beyond the initial study. This thesis wants to achieve a realistic Hardware Task Superscalar (HTSS) design, and, at the same time,

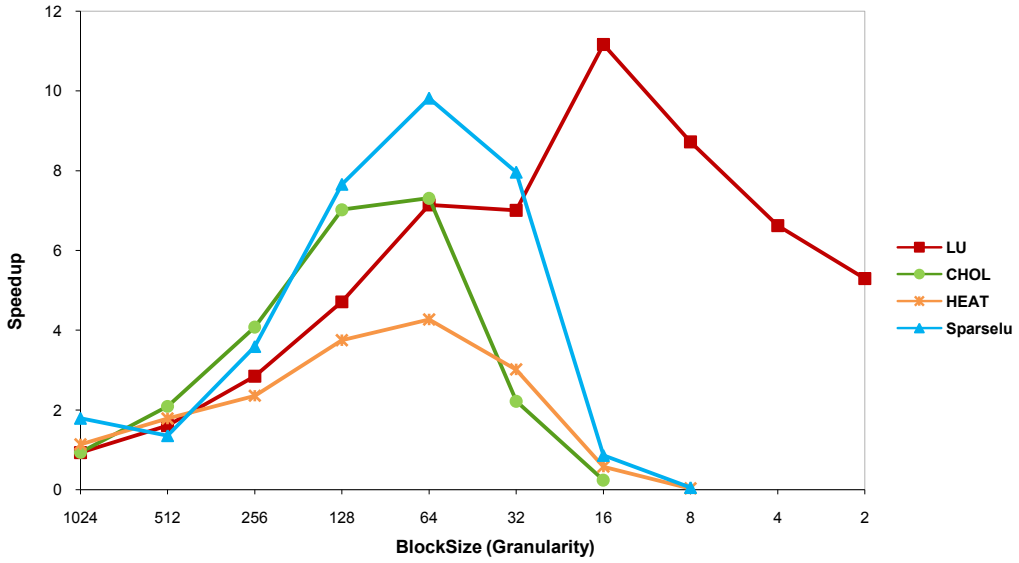


Figure 1.2: Speed-up obtained for different OmpSs applications for 2048×2048 elements in a real execution with 12 cores

provides the necessary Know-How to achieve the real implementation of the system with all the little details that arise in real-world applications.

For this, the architecture has been re-designed to be synthesizable in hardware. VHDL has been used to describe the hardware design of each of the modules which can be mapped on an FPGA. An FPGA design provides the flexibility of being easily modifiable and, at the same time, guarantees that the design would work in a future real hardware implementation.

1.1.2 Objective

The objectives of this thesis are:

- Re-design the Task Superscalar Architecture to achieve a real hardware design
- Implement a preliminary prototype in order to obtain approximate latency and throughput of the components and processes of the hardware architecture
- Create a simulator based on the hardware design implemented, configure it with the latency obtained in hardware, and perform a rapid design exploration of the

relation between components of the hardware based in real benchmarks, and compare the hardware task scheduler with the software approach (Nanos++ runtime system)

- Estimate the hardware needed for a real implementation with different design configurations, proposing HTSS configurations for HPC systems of different sizes (number of cores).

1.1.3 Contributions

- Fahimeh Yazdanpanah, Daniel Jimenez-Gonzalez, Carlos Alvarez-Martinez, Yoav Etsion, and Rosa M. Badia, *FPGA-Based Prototype of the Task Superscalar Architecture*, In HiPEAC Workshop on Reconfigurable Computing, 2013.

- Fahimeh Yazdanpanah, Daniel Jimenez-Gonzalez, Carlos Alvarez-Martinez, Yoav Etsion, and Rosa M. Badia, *Analysis of the Task Superscalar Architecture Hardware Design*, International Conference on Computational Science, ICCS 2013, 2013.

- Fahimeh Yazdanpanah, Daniel Jimenez-Gonzalez, Carlos Alvarez-Martinez and Yoav Etsion, *Hybrid Dataflow/von-Neumann Architectures*, IEEE Transaction on Parallel and Distributed System (TPDS), accepted date: 2013.

- Fahimeh Yazdanpanah, Carlos Alvarez-Martinez, Daniel Jimenez-Gonzalez, Rosa M. Badia and Mateo Valero, *Picos: A Hardware Runtime Architecture Support for OmpSs*, pre-accepted in Journal of Future Generation Computer Systems (FGCS) Elsevier, 2014.

1.2 Thesis Outline

This chapter starts with the motivation for this thesis dissertation. Then, the problem statement, the goals and the contributions of this thesis have been presented. This chapter finishes after giving an outline of the thesis. The reminder of this document is organized as follows:

Chapter 2 presents the backgrounds of our work including a brief review of the von-Neumann and dataflow computing models as well as hybrid dataflow/von-Neumann

architectures. It also explains task-based dataflow programming models, paying more attention to the StarSs family and in particular to the OmpSs programming model. Then, it describes the Task Superscalar architecture as the baseline design, pointing out its strengths and limitations. Finally, a literature survey on the related publications is presented.

Chapter 3 explains the operational flow of our proposal for hardware design of the Task Superscalar architecture, comparing it to the one of the original design. Furthermore, it explains two improved designs which are applied to the base hardware design.

Chapter 4 presents the latency exploration of the HTSS design by presenting three prototypes of the HTSS according to the three designs as well as discussing challenges of hardware prototyping. The prototypes have been written in VHDL in order to simulate them with an HDL simulator at register transfer level (RTL). Results of this exploration are required latencies for creating, processing, storing and retrieving different packets of the HTSS.

Chapter 5 describes the proposed cycle accurate simulator called SimTSS which is designed for hardware design exploration of HTSS. Then, it describes the methodology, experimental framework and real-world benchmark applications used for the design space exploration. The obtained results are presented for different HTSS configurations to determine the best HTSS configuration with the minimum number of components and the minimum memory capacity that provides maximum performance. Finally, the performance of HTSS is compared to Nanos++ runtime system.

Chapter 6 presents the hardware analysis of the designed prototypes using synthesis results. In this chapter, synthesis results of individual modules of HTSS are presented and discussed. The results are used for estimating the hardware usage of different HTSS configurations. After presenting the experimental frameworks, synthesis tools and devices that are used through this chapter, the synthesis results of the individual modules of the hardware prototypes are presented. Subsequently, the hardware resource usage of the integrated prototype, using the HTSS configurations obtained from the design space exploration, are estimated and analyzed.

This thesis study concludes in Chapter 7, by presenting some insights on the results and discussing the future plans and the possible improvements and modifications that can be done on the proposed design. It also summarizes some important points learned from this research.

Finally, there are four appendices, with supplementary tables, that include information about VHDL common signals used, characteristics of the memory modules, definition of the communication packets, and some detailed results of the SimTSS for some benchmarks.

Chapter 2

Background and Context

In this chapter, the topics addressed in the thesis are introduced: hybrid dataflow/von-Neumann architectures and task-based dataflow programming models. The Task Superscalar architecture is a hybrid dataflow/von-Neumann architecture that dynamically schedules tasks in out-of-order manner. A deep survey on hybrid dataflow/von-Neumann architectures [8] has been performed. Based on that, this chapter explains some important conclusions about von-Neumann and dataflow computing models and hybrid dataflow/von-Neumann architectures. The Task Superscalar is also a hardware task scheduler that supports the OmpSs programming model. For this reason, the main task-based dataflow programming models are explained focusing on the OmpSs programming model. After presenting the context, the Task Superscalar architecture is overviewed as the baseline of this thesis. Finally, the related work of this thesis is presented.

2.1 Von-Neumann Computing Model

The von-Neumann computing model [9] is the most common and commercially successful model to date. The main characteristic of this model is a single separate storage structure (the memory) that holds both program and data. Another important characteristic is the transfer of control between addressable instructions, using a program counter (PC). The transfer is either implicit (auto-increment of PC) or through explicit control instructions (jumps and branches, assignment to PC). It is for this reason that the von-Neumann model is commonly referred to as a control flow model.

A key tenet of the model is the set of memory semantics it provides in which loads and stores occur in the order in which the PC fetched them. Enforcing this order is required to preserve true (read-after-write), output (write-after-write), and anti (write-after-read) dependences between instructions. So, the serial execution of instructions is a hallmark of the von-Neumann architecture. However, this simplistic sequential execution, together with data, control and structural hazards during the execution of instructions, may be translated into an under-utilization of the hardware resources.

2.1.1 Parallelism in the von-Neumann Computing Model

Exploiting parallelism in the von-Neumann architecture at different granularities (i.e., instruction level parallelism (*ILP*), data level parallelism (*DLP*), and thread level parallelism (*TLP*)) is a mechanism for increasing hardware resource utilization.

Pipelined (IBM Stretch 1959 [10]) and superscalar [11] processors that try to process several instructions at the same time are the most common examples of **ILP**. Arguably the most notable class of superscalar processors is Out-of-Order processors[12] that maintain a window of pending instructions dispatching them in dataflow manner. In all these processors, parallelism is further enhanced by using a set of techniques such as register renaming, branch prediction and speculative execution, which are used in addition to dynamically dispatching independent instructions in parallel to multiple functional units. Another way of exploiting ILP is by means of very long instruction word (VLIW) processors [13]. The explicitly parallel instruction sets for VLIW enable the compiler [14] to express instruction independence statically in the binary code, thereby reducing the necessary hardware support for dynamically managing data and control hazards in Out-of-Order processors.

Architectures with **DLP** apply a single operation to multiple, independent data elements. Probably the most common examples of DLP are the single instruction multiple data (SIMD) extensions. SIMD extensions are mechanisms that statically express parallelism in the form of a single instruction that operates on wide, multi-element registers (a method sometimes referred to as sub-word parallelism). These extensions appeared in supercomputers such as the Thinking Machines CM-1 [15] and CM-2 [16], and are now ubiquitous in all general purpose processors. A derivative of SIMD processors, known as the single instruction multiple thread (SIMT) architecture, is nowadays common in graphics processing units (GPUs) [17].

TLP (or multi-threading) is applied by executing parallel threads on separate processing units. Nevertheless, some architectures utilize this coarse-grained parallelism to hide memory latencies and improve the utilization of hardware resources by interleaving multiple threads on a single physical processor. This technique is known as simultaneous multi-threading (SMT) [18, 19] and has been implemented in large machines [20, 21, 22, 23, 24]). SMT has even made it to consumer products, starting with the Pentium 4 [25] and Power 5 [26] processors. However, despite all these efforts, effective utilization of parallel von-Neumann machines is inherently thwarted by the need to synchronize data among concurrent threads. Thread synchronization and memory latencies were identified [27] as the fundamental limitations of multiprocessors.

The need for efficient data synchronization has grave programmability implications and has placed emphasis on the cache coherency and consistency in shared-memory machines, particularly as the number of processing units continuously increases [28]. Transactional memory architectures [29] aim to alleviate that problem somewhat by providing efficient and easy-to-use lock-free data synchronization. Alternatively, speculative multithreading architectures exploit TLP dynamically by scheduling the threads in parallel [30], as Out-of-Order architectures do for instructions, masking the synchronization issues. Experience shows that multithreaded control flow machines are feasible, although memory latency and synchronization may affect their scalability.

In summary, improvements in the memory system, ILP, DLP and TLP significantly reduce the memory latency issue of von-Neumann architectures, but they are still limited by the execution in control flow manner. On the other hand, the dataflow architectures can overcome this limitation due to the exploitation of the implicit parallelism of programs [27, 31].

2.2 Dataflow Computing Model

The dataflow computing model represents a radical alternative to the von-Neumann computing model. This model offers many opportunities for parallel processing, because it has neither a program counter nor a global updatable memory, i.e., the two characteristics of the von-Neumann model that inhibit parallelism. Due to these properties, it is extensively used as a concurrency model in software and as a high-level design model for hardware.

The principles of dataflow were originated by Karp and Miller [32]. They proposed a graph-theoretic model for the description and analysis of parallel computations. Shortly after, in the early 1970s, the first dataflow models were developed by Dennis [33] and Kahn [34]. Dennis originally applied the dataflow idea to the computer architecture design while Kahn used it in a theoretical context for modeling concurrent software. Based on these models, dataflow has been used and developed in many areas of computing research such as in digital signal processing, reconfigurable computing, high-level logic design, graphics processing and data warehousing. It is also relevant in many software architectures including database engine designs and concurrent computing frameworks.

The dataflow model is self-scheduled since instruction sequencing is constrained only by data dependencies. Moreover, the model is asynchronous because program execution is driven only by the availability of the operands at the inputs to the functional units. Specifically, the firing rule states that an instruction is enabled as soon as its corresponding operands are present, and executed when hardware resources are available. If several instructions become fireable at the same time, they can be executed in parallel. This simple principle provides the potential for massive parallel execution at the instruction level. Thus, dataflow architectures implicitly manage complex tasks such as processor load balancing, synchronization, and accesses to common resources.

Figure 2.1 illustrates the execution of a simple loop using both von-Neumann and dataflow models. As the figure shows, a dataflow program is represented as a directed graph, called *dataflow graph* (DFG). This consists of named nodes and arcs that represent instructions and data dependencies among instructions, respectively [35, 36]. Data values propagate along the arcs in the form of packets, called *tokens*. Two important characteristics of the dataflow graphs are functionality and composability. Functionality means that the evaluation of a graph is equivalent to the evaluation of a mathematical

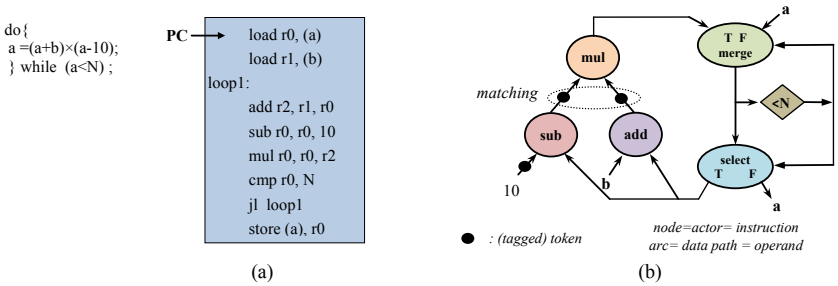


Figure 2.1: Computing a loop using (a) the von-Neumann model,(b) a dataflow model

function on the same input values. Composability implies that graphs can be combined to form new graphs [37]. A DFG can be created at different computing stages. For instance, it can be created for a specific algorithm used for designing special-purpose architectures (common for signal processing circuits). However, most dataflow-based systems convert a high-level code into DFG at compile time, decode time, or even during execution time, depending on the architecture organization. Unlike control flow programs, binaries compiled for a dataflow machine explicitly contain the data dependency information.

2.2.1 Dataflow Architectures

In practice, implementation of the dataflow model can be classified as static (single-token-per-arc) and dynamic (multiple-tagged-token-per-arc) architectures. The static approach allows at most one token to reside on any arc [38]. This is accomplished by extending the basic firing rule as follows: A node is enabled as soon as tokens are present on its input arcs and there is no token on any of its output arcs [39]. In order to implement the restriction of having at most one token per arc, and to guard against non-determinacy, extra reverse arcs carry acknowledge signals from consuming to producing nodes [39].

Figure 2.2-a shows an example of static dataflow graph for computing a loop which is executed N times sequentially (note that in this figure, the graph for controlling iteration of the loop is not illustrated). The implementation of the static dataflow model is simple, but since the graph is static, every operation can be instantiated only once, and thus loop iterations and subprogram invocations can not proceed in parallel. Despite this drawback, some machines were designed based on this model, including the MIT Dataflow Architecture [38, 40], DDM1 [41], LAU [42], and HDFM [43].

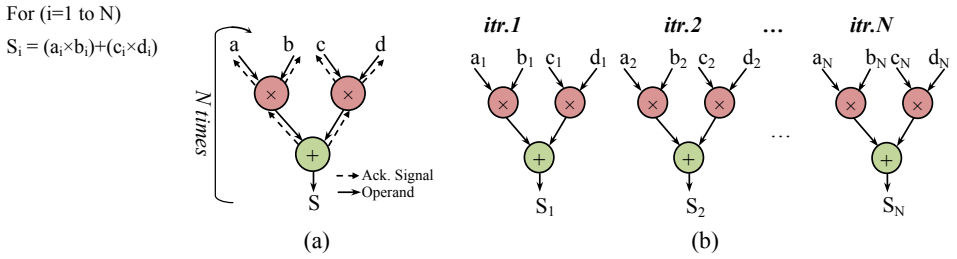


Figure 2.2: DFG of a loop (a) the static and (b) the dynamic dataflow

The dynamic dataflow model tries to overcome some of the deficiencies of static dataflow by supporting the execution of multiple instances of the same instruction template, thereby supporting parallel invocations of loop iterations and subprogram. Figure 2.2-b shows the concurrent execution of different iterations of the loop. This is achieved by assigning a *tag* to each data token representing the dynamic instance of the target instruction (e.g., a_1, a_2, ...). Thus, an instruction is fired as soon as tokens with identical tags are present at each of its input arcs. This enabling rule also eliminates the need for acknowledge signals, increases parallelism, and reduces token traffic. Notable examples of this model are the Manchester Dataflow [44], the MIT Tagged-Token [45], DDDP [46] and PIM-D [47].

The dynamic dataflow can execute out-of-order, bypassing any token with complex execution and delays the remaining computation. Another noteworthy benefit of this model is that little care is required to ensure that tokens remain in order.

The main disadvantage of the dynamic model is the extra overhead required to match tags on tokens. In order to reduce the execution time overhead of matching tokens, dynamic dataflow machines require expensive associative memory implementations [44]. One notable attempt to eliminate the overheads associated with the token store is the *Explicit Token Store* (ETS) [48, 49]. The idea is to allocate a separate memory frame for every active loop iteration and subprogram invocation. Since frame slots are accessed using offsets relative to a frame pointer, the associative search is eliminated. To make that concept practical, the number of concurrently active loop iterations must be controlled. Hence, the condition constraint of k-bounded loops was proposed [50], which bounds the number of concurrently active loop iterations. The Monsoon architecture [51] is the main example of this model.

2.2.2 Limitations of Dataflow Models

The dataflow model has the potential to be an elegant execution paradigm with the ability to exploit inherent parallelism available in applications. However, implementations of the model have failed to deliver the promised performance due to inherent inefficiencies and limitations. One reason for this is that the static dataflow is unable to effectively uncover large amount of parallelism in typical programs. Dynamic dataflow architectures are limited by prohibitive costs linked to associative tag lookups, in terms of latency, silicon area, and power consumption.

Another significant problem is that dataflow architectures are notoriously difficult to program because they rely on specialized dataflow and functional languages. However, these languages have no notion of explicit computation state, which limits the ability to manage data structures (e.g., arrays). To overcome these limitations, some dataflow systems include specialized storage mechanisms, such as the I-structure [52], which preserve the single assignment property. Nevertheless, these storage structures are far from generic and their dynamic management complicates the design.

In contrast, imperative languages such as C, C++, or Java explicitly manage machine state through load/store operations. This modus operandi decouples the data storage from its producers and consumers, thereby concealing the flow of data and making it virtually impossible to generate effective (large) dataflow graphs. Furthermore, the memory semantics of C and C++ support arithmetic operations on memory pointers, which result in memory aliasing, where different semantic names may refer to the same memory location. Memory aliasing cannot be resolved statically, thus further obfuscating the flow of data from between producers and consumers. Consequently, dataflow architectures do not effectively support imperative languages.

In summary, the dataflow model is effective in uncovering parallelism, due to the explicit expression of parallelism among dataflow paths and the decentralized execution model that obviates the need for a program counter to control instruction execution. Despite these advantages, programmability issues limit the usefulness of dataflow machines. Moreover, the lack of a total order on instruction execution makes it difficult to enforce the memory ordering that imperative languages require. For further details, we refer the reader to more extensive literature on the subject [53, 54, 55].

2.3 Hybrid Dataflow/von-Neumann Architectures

The inherent limitations of both dataflow and von-Neumann execution models motivate the exploration of a convergent model that can use synergies to leverage the benefits of both individual models. Therefore, the hybrid models try to harness the parallelism and data synchronization inherent to dataflow models, while maintaining existing programming methodology and abstractions that are largely based on von-Neumann models. While different hybrid implementations differ in the way they merge the two conceptually different models, they all follow similar principles.

Most notably, hybrid models alleviate the inefficiencies associated with dataflow model, either by increasing the basic operation granularity or by limiting the size of the DFG. Additionally, they incorporate control flow abstractions and shared data structures. As a result, different hybrid architectures employ a mix of control flow and dataflow instruction scheduling techniques using different partial scheduling methods. Furthermore, in the hybrid models, nodes of a DFG vary between a single instruction (fine-grained) to a set of instructions (coarse-grained).

A further significant benefit of hybrid models is clearly evident in their memory models. Hybrid models combine single assignment semantics, inherent to dataflow, with consistent memory models that support external side-effects in the form of load/store operations. This relieves one of the biggest (if not the biggest) restriction of pure dataflow programming: the inability to support a shared state, and specifically shared data structures [55]. Therefore, hybrid models are capable of executing imperative languages. As a result, combining dataflow and von-Neumann models facilitates designing efficient architectures that benefit from both computing models, while the remaining issue concerns the best granularity-parallelism trade-off.

2.3.1 Evolution of Hybrid Architectures

The first idea of combining dataflow and control flow arose in the early 1980s [20, 56, 57, 58], and included data and memory structure management [59], self-scheduling and asynchronous execution to simplify thread synchronization [20, 21, 57, 59], as well as the ability to execute both conventional and dataflow programs in the same machine [27, 28]. Some hybrid models [28, 60] even included a program counter to a dataflow architecture in order to execute sequential instructions in control flow manner.

In this regard, other studies explored the *threaded dataflow* model [37, 61], in which partial data sub-graphs are processed as von-Neumann instruction streams. In particular, given a dataflow graph (program), each sub-graph that exhibits a low degree of parallelism is identified and transformed, into a sequential thread of instructions. Such a thread is issued consecutively by the matching unit without matching further tokens, except for the first instruction of the thread. Data passed between instructions in the same thread is stored in registers instead of being written back to memory. These registers may be referenced by any succeeding instruction in the thread. This improves single-thread performance, because the total number of tokens needed to schedule program instructions is reduced, which in turn saves hardware resources. In addition, pipeline bubbles caused by runtime overhead associated with token matching are avoided for dyadic (two-operand) instructions within a thread. Two threaded dataflow execution techniques can be distinguished: (1) the direct token recycling technique, which allows cycle-by-cycle instruction interleaving of threads in a manner similar to multithreaded von-Neumann computers (e.g., MT. Monsoon architecture), and (2) consecutive execution of the instructions of a single thread technique (e.g., Epsilon [62, 63] and EM-4 [64] architectures). In the second technique, the matching unit is enhanced with a mechanism that, after firing the first instruction of a thread, delays matching of further tokens in favor of consecutive issuing of all instructions of the started thread. In addition, some architectures based on threaded dataflow use instruction pre-fetching and token pre-matching to reduce idle times caused by unsuccessful matches. EM-4 [64], EM-X [65] and RWC-1 [66] are examples of this kind of architectures, which are also referred to as *macro-dataflow* [67].

Until the early 90s, the common wisdom was that fine-grained execution was much more suited to masking network and memory latencies than a coarse-grained execution, and would obviously provide a much better load leveling across processors and hence faster execution. However, it has been demonstrated that coarse-grained execution [68, 69, 70, 71, 72, 73, 74, 75, 76, 77] is equally suited to exploit parallelism as fine-grained.

In addition to the coarsening of nodes in the DFG, another technique for reducing dataflow synchronization frequency (and overhead) is the use of complex machine instructions, such as vector instructions. With these instructions, structured data is referenced in block rather than element-wise, and can be supplied in bursts while also

introducing the ability to exploit parallelism at the sub-instruction level. This technique introduces another major difference with conventional dataflow architectures; that is, tokens do not carry data (except for the values `true` or `false`). Data is only moved and transformed within the execution stage. Examples of such machines are Stollman [78], ASTOR [79], DGC [74, 75], and SIGMA-1 multiprocessor [80].

In parallel, the Out-of-Order model [12, 81], which emerged in the late 80s, incorporated the dataflow model to extract ILP from sequential code. This approach has been further developed by Multiscalar [30] and thread level speculation (TLS) [82, 83], which can be viewed as coarse-grained versions of Out-of-Order.

Efforts have been made to survey hybrid models up to year 2000 [37, 61, 84], and also dataflow multithread models [67, 85, 86, 87]. Yazdanpanah et. al. [8] present a survey on hybrid dataflow/von-Neumann architectures, which has mainly attempted to improve the conventional architectures exploiting several aspects of dataflow concepts [7, 88, 89, 90, 91, 92, 93, 94, 95], or to utilize the dataflow approach as accelerators [96, 97, 98, 99].

2.3.2 Taxonomies of Hybrid Models

The inherent differences between dataflow and von-Neumann execution models appear to place them at two ends of a spectrum that covers a wide variety of hybrid models. However, the coarsening of the basic operation granularity, from a single instruction to a block of instructions, together with the inter- and intra-block execution semantics, enable the partition of the spectrum into four different classes of hybrid dataflow/von-Neumann: *Enhanced Control Flow*, *Control Flow/Dataflow*, *Dataflow/Control Flow* and *Enhanced Dataflow* class. This taxonomy is based on whether they employ dataflow scheduling between and/or inside code blocks. Block is defined on the basis of the boundary between where the two scheduling models (inter- and intra-block scheduling) are mainly applied. In this way, the number of instructions in a block (block granularity) depends on the specific model. Figure 2.3 illustrates inter- and intra-block scheduling of conventional organizations of hybrid dataflow/von-Neumann architectures.

Models in *Enhanced Control Flow* Class schedule blocks in control flow manner, whereas the instructions within a block are scheduled in a mixed approach of control flow and dataflow manner. Figure 2.3-a) illustrates the organization of this class. The main example of this class is the Out-of-Order (restricted dataflow) model [12, 81].

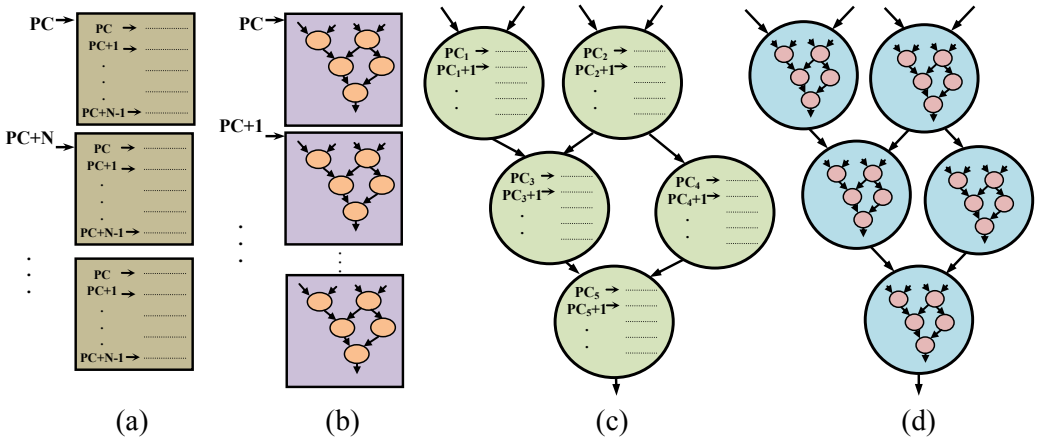


Figure 2.3: Inter- and intra-block scheduling of organizations of hybrid dataflow/von-Neumann architectures. (a) Enhanced Control Flow, (b) Control Flow/Dataflow, (c) Dataflow/Control Flow, and (d) Enhanced Dataflow. Blocks are squares and big circles

Enhanced Control Flow class machines can very naturally execute control flow codes and uncover more ILP than the strict von-Neumann models. However, as the technology only allows them to address small to medium block sizes, the amount of parallelism they can expose is typically limited (some architectures such as Kilo-instruction Processors [100] try to overcome this problem by targeting much larger block sizes).

Models in *Control Flow/Dataflow* Class schedule the instructions within a block in dataflow manner, whereas blocks are scheduled in control flow manner (Figure 2.3-b). This method is used in RISC dataflow architectures, which support the execution of existing software written for conventional processors. Main examples of this class are TRIPS [88, 89], Tartan [96], Conservation Cores (C-Cores) [97], DySER [98] and other architectures that rely on domain specific dataflow accelerators. *Control Flow/Dataflow class* machines try to overcome the limitations of the previous class by forcing the pure dataflow execution of the instructions inside a block. These models attempt to expose ILP statically at the block level, deferring memory operations to inter-block synchronization. Indeed, the *Control Flow/Dataflow* general strategy has shown a great potential in both performance and power savings [96, 98], although it poses the same problems as the previous class (e.g., smaller block sizes than desirable for fully exploiting dataflow advantages at ILP level).

Models in *Dataflow/Control Flow* class employ dataflow rules between blocks and control flow scheduling inside the blocks (Figure 2.3-c). A block is a set of sequen-

tial instructions, where data is passed between instructions using register or memory (coarse-grained dataflow models [37, 61, 84]). Under these restrictions, blocks are issued by the matching unit, and token matching needs only to be performed on a block basis. Thus, the total number of tokens needed to schedule program instructions is reduced, which in turn saves hardware resources. Main examples of this class are: Star-T (*T) [101], TAM [102], ADARC [103], EARTH [73, 104], P-RISC [105], MT. Monsoon [59], Pebbles [77], SDF[1] [92], DDM [95], and Task Superscalar (TSS) [7]. The models of this class have taken advantage of the recent growth in the number of parallel hardware structures in cores, chips, machines and systems. As models in this class address parallelism at a coarse grain, they are able to exploit all these resources more effectively than conventional (von-Neumann) models while retaining the programming model inside the blocks. As Task Superscalar architecture is a member of this class, we will explain this class more in detail in the following section.

Models in *Enhanced Dataflow* Class use dataflow firing rules for instructions inside the blocks and for the blocks themselves. In effect, this class consists of two-level dataflow models (Figure 2.3-d) utilizing some concepts of the von-Neumann model (e.g., storage management) to add the abilities of running imperative languages and managing data structures. SIGMA-1 [80], Cedar [107] and WaveScalar [90] are the main examples in this class. *Enhanced Dataflow* models constitute a complete re-thinking of the execution problem. Since they do not use a program counter, they face several difficulties when executing conventional codes and managing memory organizations, and therefore need more hardware resources to be used effectively. On the other hand, *Enhanced Dataflow class* models may be regarded as an addition to both *Dataflow/Control Flow* and *Control Flow/Dataflow* classes, and in this sense they posse great potential.

Hybrid models can also be classified from an execution model point of view; *unified-hybrid* models versus *dataflow accelerator* models. In a unified-hybrid architecture, a program must be executed using both dataflow and control flow scheduling since both models are intimately bound in the architecture. Although the majority of the models presented belong to this group, it does present some drawbacks. The additional hardware needed by the interconnection and synchronization mechanisms (e.g., hardware

[1] Please note that here SDF is the acronym for scheduled dataflow, as opposed to synchronous dataflow (SDF) [106]. The latter is a dataflow based execution model for signal processing algorithms and does not include any von-Neumann properties.

of Out-of-Order architectures) leads to more complexity and power consumption. Furthermore, as all programs should be executed with the same hybrid scheduling schema, they are not able to adapt to specific cases in which a pure dataflow or von-Neumann model would be better.

On the other hand, in architectures with dataflow accelerators, the decision about which parts of the code to accelerate is mostly static (made by the programmer or compiler, and sometimes based on profiling). In addition, a whole program may be executed without the use of dataflow accelerators. Tartan, C-Cores and DySER are architectures that use dataflow to accelerate kernels (or hyperblocks).

2.3.3 Models of Dataflow/Control Flow Class

In *Dataflow/Control Flow* architectures, blocks are scheduled in a dataflow manner, while control flow scheduling is used within the blocks. Figure 2.4 shows a further decomposition of this class based on the number of cores and number of instructions in a block (i.e., size of block) targeted by every specific model, as well as the year in which it was first published. Figure 2.4-a depicts the relationship between core granularity and the publication year of the proposed architectures. First hybrid designs tend to have a small number of cores, while recently proposed architectures tend to use a larger number of cores. Figure 2.4-b shows the variance in core granularity in hybrid design. Architectures with a larger number of cores typically use fewer numbers of instructions per block, and designs with a fewer number of cores tend to use larger blocks (with more than 1000 instructions per block).

Table 2.1 introduces the main features of the main representations of this class sorted according to the *year* in which the architecture appeared.

DDM and Task Superscalar are based on RISC/CISC ISA. SDF is based on a RISC ISA defined for the execution and synchronization processors. MT. Monsoon is based on dataflow ISAs. The main features of MT. Monsoon architecture are the Explicit Token Store (ETS), which eliminates the associative search in the matching unit, and multithreading. The main feature of DDM is the introduction of the CacheFlow policy, which implies the execution of a DDM thread (basic block of instructions - BB) only if its data is already placed in the cache. Decoupling computation and synchronization, and non-blocking threads are also the main features of SDF and DDM. However, computation in the DDM is carried out by an off-the-shelf processor, while in the SDF it is

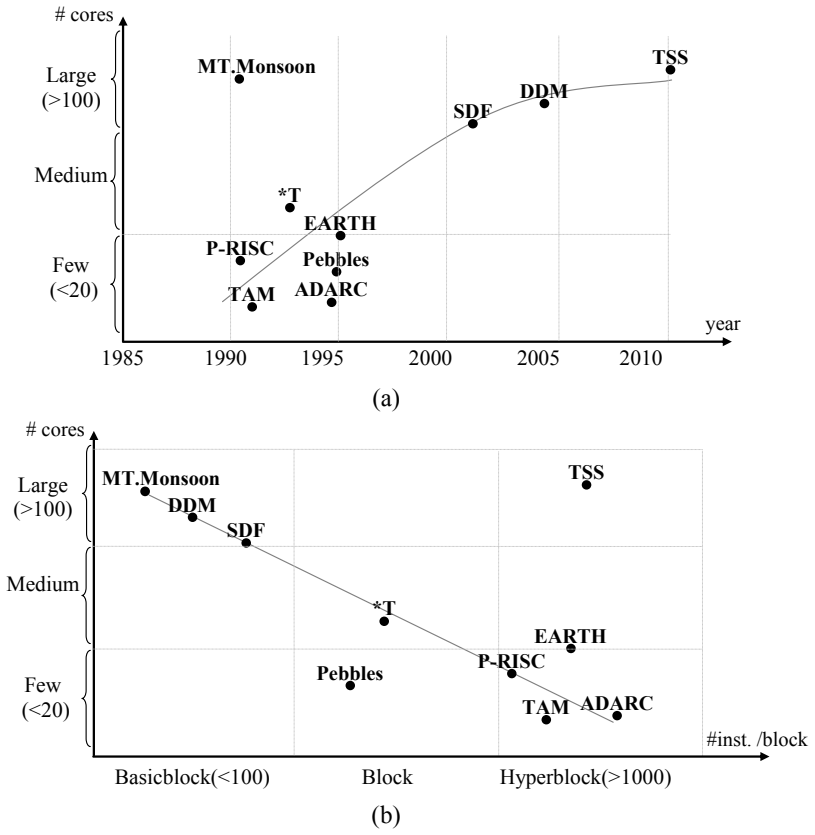


Figure 2.4: Different architectures of Dataflow/Control Flow class (a) number of cores and year, (b) number of cores and size of blocks

carried out by a custom designed processor. Another difference is that in SDF data is preloaded in registers, while in DDM data is pre-fetched in the cache. The main feature of Task Superscalar is out-of-order task execution. The computational core granularity varies from any processing element (PE) or core size in the case of DDM and Task Superscalar to a small SDF core. MT. Monsoon uses the original dataflow Monsoon PE to sequentially execute the thread instructions using the direct token recycling technique.

Models in this class tend to provide specific support only to TLP. In particular, based on dependencies specified in the program, DDM and Task Superscalar perform dynamic dataflow inter-block scheduling by using cache and memory, respectively, for inter-block communication. SDF and MT. Monsoon perform static dataflow and both use memory and registers for inter-block communication. Blocks of DDM, SDF and MT. Monsoon are equivalent to a basic block, being up to 128 instructions in the case

Table 2.1: Comparison of the hybrid dataflow/von-Neumann architectures in the class of dataflow/control flow class. DF, CF, and DFG stand for dataflow, control flow and dataflow graph, respectively

Architecture	MT. Monsoon	DDM	SDF	Task Superscalar
Year	1991	2000	2001	2010
ISA	Dataflow graph w/ thread extensions	RISC / CISC	RISC (Preload/ store + computation)	RISC / CISC
Main features	ETS, MT	Decoupled non-blocking, CacheFlow policy (prefetching)	Decoupled non-blocking multithreading	Out-of-order task execution
Core Granularity	Monsoon PE	PE agnostic	Simple processor	PE agnostic
Scalability	> 1000PEs	~ 100 PEs	~ 100 PEs	>> 100 PEs
Parallelism level	TLP	TLP	TLP	TLP
Block Granularity	Thread (BB <=128)	BB size (code block, more than one thread, in TSU graph memory)	BB size, <128-inst. blocks (27 (15 in EP) up to 51 (39 in EP))	Task size (any size) > 10K
Inter-block Scheduling	Dataflow	Dynamic dataflow (dependencies specified in programs)	Static dataflow (programmer/ compiler)	Dynamic dataflow (dependencies specified in programs)
Intra-block Scheduling	Static control flow (thread sequential execution)	Control flow	Control flow (scheduled dataflow)	Control Flow
Inter-block Communication	Register / memory	Cache	Frame memory and registers	Memory
Intra-block Communication	Register / memory	Register / memory	Register	Register / memory
Examples	MT. Monsoon	D2NOW, Flux, DDM-VMc	SDA	Task Superscalar

of a SDF/MT. Monsoon block. Task Superscalar may have blocks of any size.

The sizes of blocks of DDM, SDF, and MT. Monsoon model tend to be small, a decision that allows large amount of parallelism to be discovered and executed but also increases the cost of the synchronization. In the case of DDM, this characteristic makes the thread scheduling unit as important as the workstation duplicating the number of necessary processing elements. Another key point in this model is that in order to be efficient, it needs more information about the program than the classical control flow model. Programs should thus be annotated either by the compiler or by the programmer, which increases the complexity of the tool-chain needed to develop new applications. Unlike DDM, SDF executes the instructions within a block in-order, thereby obtaining less ILP but allowing the execute processor of its architecture to be simpler and smaller. Another characteristic of the SDF paradigm is that, although it can benefit from the annotated code, it can execute the original code as is, automatically extracting the available parallelism. MT. Monsoon, however, executes instructions within

a block in-order using the direct token recycling technique. In addition, the thread extensions included in the MT. Monsoon facilitate the fork, join, and split phases of block executions.

In the Task Superscalar, blocks are designed to be as large as desired. The Task Superscalar pipeline is designed as a generalization of Out-of-Order processors to the task-level. Nevertheless, its scalability goals, which target dynamically, managing very large graphs consisting of tens of thousands of nodes, require an alternative design to that of Out-of-Order processors. This re-design is the result of the Out-of-Order pipeline's use of reservation stations and bypass networks, whose operation is similar to that of associative token stores and are known not to scale.

The main disadvantage of most of models in this class is the need for annotating the original codes in order to extract a significant amount of parallelism from these codes. In this sense, a trend towards simplifying the annotations as much as possible can be observed in the designs of the programming models.

2.4 Task-based Dataflow Programming Model

Most of parallel programming models such as Cilk [108], OpenMP [109], Intel TBB [110], CUDA [111] and OpenCL [112] burden the programmer with the non-trivial assignment of resolving inter-task data dependencies. In addition, many task-based programming models use (possibly partial) barrier synchronization, which inhibits utilization of distant parallelism. Task-based dataflow programming models do not have these drawbacks because they automate data dependency resolution. In these models, task definition is explicit, so it is needed for the programmer to specify them. These models use programmer annotations to dynamically construct the inter-task data dependency graph and extract task parallelism at runtime. In addition, they implicitly manage parallelism, so the programmers do not need to care about that management in the program. In these programming models, the programmer is not responsible for exposing the structure of the task graph. Instead, the task graph is built automatically, based on the information of task dependences and their directionality.

Main examples of task-based dataflow programming models are Jade [4, 113], StarSs [2, 3, 114], Sequoia [115], Intel RapidMind [116, 117, 118], OoOJava [5, 119] and the recently appeared OpenMP 4.0 [120]. In the following section, we introduce StarSs programming family in order to describe the principle of task-based dataflow programming models.

2.4.1 StarSs Programming Family

The Star Superscalar (StarSs) [2, 3, 114] is a family of task-based dataflow programming models that support dataflow and out-of-order execution of tasks. StarSs implementations analyze dependencies among tasks and manages their execution, exploiting as much as possible their parallelism.

Each member of the StarSs family targets a particular architecture; SMPSs [2] for symmetric multiprocessors, CellSs [3] for the Cell/BE processor, GPUSs for graphics accelerators, GRIDSs for Grids architectures and ClusterSs [121, 122] for clustered architectures. OmpSs [123, 124] joins the capabilities of several previous implementations to cover different architectures (like SMPs, GPUs anf FPGAs) with an homogeneous model. Nevertheless, all of them share the same philosophy: the user is required to annotate a set of instructions of a sequential application that will run as a task on the

available resources. For each task, we need to specify its dependences and the direction of the dependences (i.e.,input, output or inout). This information is used by StarSs programming models to discover, at execution time, the data dependencies between tasks and exploit the implicit parallelism among the application tasks.

StarSs programming model offer a thread-pool execution model where threads of this pool can create tasks that are executed by the idle threads of this pool. StarSs programming models assume a non-homogeneous disjoint memory address space, therefore tasks must specify their data requirements in order to access them correctly. As explained, these data specifications are also used to compute the dependences between tasks, which are needed to schedule them in dataflow manner.

The sequential code flow of StarSs, together with the implicit synchronizations and data transfers provided by the runtime, guarantees that the results of a parallel execution will be equivalent to a sequential one. The runtime system also manages data transfers between main memory and local scratchpad memories, if applicable. It tries to minimize the execution time by applying few optimizations. First, the runtime system creates groups of tasks, referred to as bundles within StarSs. Using bundles, reduces the overhead per task for scheduling. In addition, the runtime system optimizes for data locality by assigning chains within the task graph to bundles. Within such bundles, data produced by one task is used by a next, and thus locality is exploited.

There are many works that have been performed in regard to StarSs. For instance, the work of Planas et. al [125] is an extension of the StarSs syntax to support task hierarchy by SMPSs and CellSs programming models with nested parallelism support. Hybrid MPI/SMPSs [126] programming model was proposed for providing asynchronous parallelism on clusters. The tool can be adapted to support similar programming models such as Tagged Procedure Calls (TPC) [127].

OmpSs Programming Model

OmpSs [123, 124] is a task-based dataflow programming model that joins the advantages of both OpenMP and StarSs. It was designed at Barcelona Supercomputing Center (BSC) and implemented in software through the Mercurium [128] source-to-source compiler and the Nanos++ runtime system.

OmpSs executes task-based parallel applications making sure all constraints specified by the programmer are maintained. Master thread starts creating new tasks. Before

a task is executed, it has to go through the dependence graph, which ensures that a task can be executed correctly. Next, the task scheduler decides in which resource (processor, node, GPU and FPGA) is going to be run. The library moves the data required by tasks among the different address spaces (nodes, GPUs or FPGAs) present in the execution. In order to achieve good scalability the runtime implements techniques like data conscious scheduling, and communication and computation overlapping; all of this being completely transparent to the user.

The OmpSs execution model is a thread-pool model. The master thread starts the execution and all other threads cooperate executing the work it creates (whether it is from work sharing or task constructs). Therefore, there is no need for a parallel region. Nesting of constructs allows other threads to become work generators (masters) as well.

Similar to other StarSs members, OmpSs allows annotating function declarations or definitions with a task directive. In this case, any call to the function creates a new task that will execute the function body. The data environment of the task will be captured from the function arguments. In OmpSs, programs are parallelized by annotating functions as tasks using `omp` pragmas. When these functions are called, they are added to a task graph instead of directly being executed. The task dependencies are resolved at runtime, using the input/output specification of the function arguments. Once all input dependencies of a task are resolved, the task is ready to be executed[1].

OmpSs assumes a non-homogeneous disjoint memory address space. As such shared data may reside in memory locations that are not directly accessible from some of the computational resources. Therefore, all parallel code can only safely access private data and for shared data it must specify how it is going to be used. This assumption is true even for SMP machines as the implementation may reallocate shared data taking into account memory effects (e.g., NUMA). In order to support heterogeneity and data motion between address spaces, OmpSs uses a new construct called `target` construct. The `target` construct can be applied to either task, work sharing constructs or functions. It allows to specify on which devices should be targeting the construct (e.g., CellBE, GPU, SMP, FPGA, etc.).

As an example, Figure 2.5 shows the sequential code (in black text) and dependency graph of the Cholesky algorithm. The tasks and their dependences are defined by the

[1]The runtime of OmpSs also guarantees that output dependences of the task do not break anti and output dependencies against other tasks.

programmer using OmpSs pragmas. The OmpSs pragmas (in bordered, dark-blue text) are inserted for each function defining the input and output dependences. As the figure shows, Cholesly has four different tasks: `spotrf` with one *inout* dependence; `strsm` with one *input* dependence and one *inout* dependence; `sgemm` with two *input* dependences and one one *inout* dependence and `ssyrk` with one *input* dependence and one *inout* dependence. The Nanos++ runtime system extracts the dependency graph from the program using the pragmas, and schedules tasks in dataflow manner. In practice, the whole dependency graph may never exist in this complete form, at runtime, because tasks appear only after they have been created, and are removed once they have been executed.

```
#pragma omp task inout ([TS][TS]A)
void spotrf (float *A);
#pragma omp task input ([TS][TS]T) inout ([TS][TS]B)
void strsm (float *T, float *B);
#pragma omp task input ([TS][TS]A, [TS][TS]B) inout ([TS][TS]C)
void sgemm (float *A, float *B, float *C);
#pragma omp task input ([TS][TS]A) inout ([TS][TS]C)
void ssyrk (float *A, float *C);
void Cholesky (int NT, float *A) {
int i, j, k;
for (k=0; k<NT; k++){
    spotrf(A[k*NT+k])
    for (i=k+1; i<NT; i++)
        strsm(A[k*NT+k], A[k*NT+i]);
    // update trailing submatrix
    for (i=k+1; i<NT; i++){
        for (j=k+1; j<i; j++)
            sgemm (A[k*NT+i], A[k*NT+j], A[j*NT+i]);
        ssyrk(A[k*NT+i], A[i*NT+i]);    }  }
```

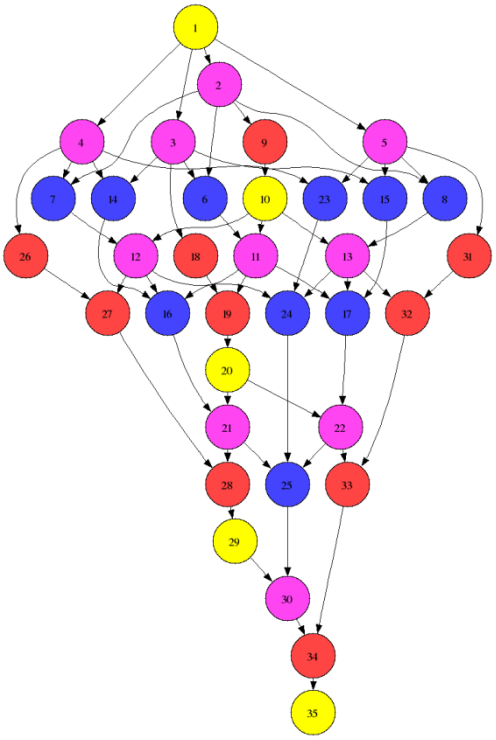


Figure 2.5: OmpSs implementation of the Cholesky algorithm and its dependency graph

2.5 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 which generalizes the operational flow of dynamically scheduled Out-of-Order processors. It was designed at the Barcelona Supercomputing Center (BSC) and has been proposed as hybrid dataflow/von-Neumann architecture [8] to support the OmpSs programming model for task scheduling and dependency analysis, belonging to the *Dataflow/Control Flow class* (see Section 2.3.3). It implements in hardware the task dependency management and task scheduling functionalities of Nanos++ runtime system thus reducing the per-task overhead; allowing the efficient exploitation of parallelism at a finer granularity. In this sense, the Task Superscalar processor combines dataflow execution of tasks with control flow execution within the tasks. The idea behind this behavior is that Task Superscalar uncovers task level parallelism among tasks generated by a sequential thread similarly as ILP pipelines uncover parallelism in a sequential instruction stream. So, the Task Superscalar combines the effectiveness of Out-of-Order processors in uncovering parallelism together with the task abstraction, thereby providing a unified management layer for CMPs which effectively employs processors as functional units.

The high-level organization of the Task Superscalar is illustrated in Figure 2.6. A task generator thread resolves the inter-task control path and sends non-speculative tasks to the pipeline front-end for dependency decoding. The task window may consist of tens of thousands of tasks, which enables it to uncover large amounts of parallelism [6]. The front-end asynchronously decodes the task dependencies, generates the task dependency graph (with tasks as nodes and dependencies between tasks as arcs), and schedules tasks as they become ready. Finally, ready tasks are sent to the execution back-end, which consists of a task scheduler and a queuing system.

As shown in Figure 2.6, the front-end employs a tiled design, and is managed by an asynchronous point-to-point protocol. The front-end is composed of five module types: the *pipeline gateway* (GW); *task reservation station* (TRS); *object renaming table* (ORT); *object versioning table* (OVT) and the task scheduler (TS).

The gateway is responsible for controlling the flow of tasks into the pipeline and distributing tasks to the different TRS modules, sending dependences to their assigned

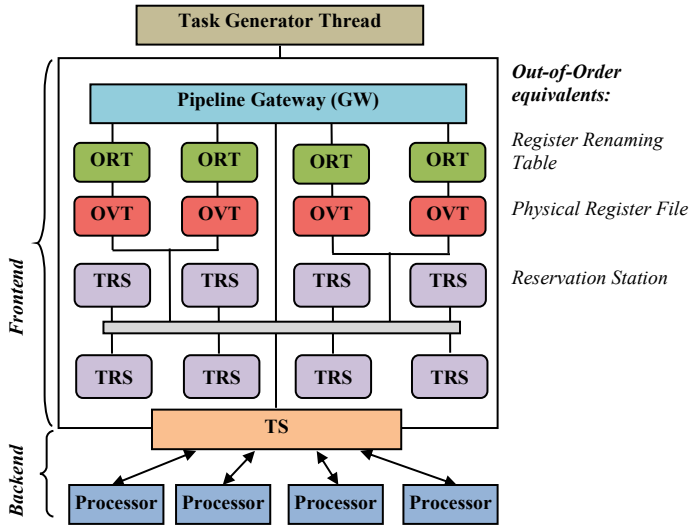


Figure 2.6: The Task Superscalar architecture (Figure based on [6])

ORT modules and stalling the task generator thread whenever the pipeline fills. TRSs store the in-flight task information and track the readiness of task operands. Inter-TRSs communication is used to register consumers with producers, and notify consumers when data is ready. The TRSs store the meta-data of all in-flight tasks (i.e., tasks waiting for their parameters (dependences) to be ready, ready tasks waiting to be sent for execution, tasks being executed and finished tasks still not retired).

The ORTs map dependences that access the same memory object, and thereby detect task dependencies. Storing data users instead of real data allows the system to maintain the dependence chain with realistic memory sizes. To keep this chain, the OVTs track live operand versions, which are created whenever a new data producer is decoded. Each OVT is associated with exactly one ORT. The functionality of the OVTs is similar to a physical register file, but only for maintaining operand meta-data. Effectively, the OVT manages data anti- and output-dependencies, either through operand renaming or by chaining different output operands and unblocking them in-order by sending a ready message when the previous version is released.

Figure 2.6 also shows, on the right, the Out-of-Order components equivalent to the Task Superscalar modules. In Out-of-Order processors, dynamic data dependencies are detected by matching each input register of a newly fetched instruction (consumer) with the most recent instruction that writes data to that register (producer). The

instruction is sent to a reservation station to wait until all its inputs become available. Hence, the reservation stations effectively store the instruction dependency graph, which consists of all in-flight instructions. In the Task Superscalar, the mechanism of decoding tasks identifies all possible effects a task may have on the shared processor state, so producers and consumers are identified correctly. Moreover, tasks are decoded in-order to guarantee correct ordering of producers and consumers, and specifically, that the decoding of a task producing a datum updates the renaming table, before any task consuming the datum performs a lookup.

The designers of the Task Superscalar opted for a distributed structure that through a careful protocol design that ubiquitously employs 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 at 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 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 scalability of the design.

2.6 Related Work

Static task management systems are not able to adapt to the variable behavior of modern algorithms [129] due to the blocking nature of synchronization. To overcome this problem, different dynamic software task management systems (i.e., runtime systems) have been proposed. In particular, an emerging class of task parallel programming models, referred as task-based dataflow programming models, automates data dependency and solves the synchronization problem of static task management systems. Main examples of this class are StarSs family [1, 2, 3] (including OmpSs [123, 124]) , OoO-Java [5, 119], JADE [4, 113, 130, 131] and OpenMP 4.0 [120]. These models try to support dynamic task 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 [114]. The cost of that potentially huge task management affects the scalability and performance of such systems, and limits their applicability to applications with a large number of tasks.

The main purpose of a hardware task scheduler is accelerating the task management. The parallel program will continue calling the programming model software runtime, but this will subministrate the task dependency information to the hardware task scheduler. The hardware scheduler will gradually and efficiently create the task dependency graph while preparing the ready tasks for execution on the cores. The threads of the runtime thread-pool continuously ask for new tasks and inform the hardware task scheduler when tasks finish. Some hardware support solutions for task scheduling [132, 133, 134, 135, 136, 137, 138] 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.

Several research studies evaluated hardware task queues, for example, the Intel CARBON [132] and Task Scheduling Unit [133]. In these studies task submission to a particular core is accelerated by hardware task queues, replacing software data structures and leveraging the corresponding synchronization overhead. However, in most of the cases inter-task synchronization is still performed in software.

There exist solutions in hardware to accelerate the synchronization between tasks, instead of relying on memory-based instructions such as LL/SC. For example, Cell Broadband Engine introduces hardware mailboxes and semaphores. The drawback of

most hardware solutions for synchronization is the restricted number of simultaneously active synchronization primitives or their limited domain applicability. Saez et al. [139] describe a hardware scheduler accelerator that combines scheduling of soft and hard real-time jobs on an uni-processor. In contrast, Al-Kadi and Terechoko [129] propose a video scheduler that tackles the task scheduling problem for a multi-core, involving complex task-to-core mapping. Furthermore, their task scheduler can create task dependency graphs, whereas the scheduler proposed by Saez et al. [139] obtains a set of independent tasks from the program. Other architectures, such as NVIDIA Tesla [138], are also known to provide hardware acceleration for task scheduling of independent tasks. Sjalander et al [134] propose a programmable task management unit (TMU), which similarly to the video scheduler of Al-Kadi and Terechoko [129] accelerates task creation and synchronization in hardware. TMU runs a look-ahead program preparing tasks that will become ready in a short while. However, for fine-grain tasks executing for a few tens of cycles, it is needed a faster task scheduling. In order to achieve lower overhead, in this thesis it is designed a dedicated hardware task scheduler to efficiently manage the task creation, scheduling, mapping and synchronization of the tasks.

Dynamic scheduling for system-on-chip (SoC) with dynamically reconfigurable architectures is interesting for the emerging range of applications with dynamic behavior. Kalra and Lyseeky [140] addressed the relationship between the several hardware task scheduling algorithms and their impact on the number of reconfigurations required to execute. As an instance, Noguera and Badia [141, 142] 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 OmpSs programming model [123] for scheduling all dependent and independent tasks. Unlike 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 parallelism management through a dynamic dataflow execution model. In addition, it supports imperative programming on large-scale CMPs without any fundamental changes to the micro-architecture. Nexus++ [143, 144] 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.

2.7 Summary

This chapter covers relevant topics to Task Superscalar architecture which is the baseline of the thesis. We presented an overview of von-Neumann and dataflow computing models as well as hybrid dataflow/von-Neumann architectures. Furthermore, we overviewed task-based dataflow programming models focusing on the OmpSs programming model since Tasks Superscalar architecture can act as a hardware scheduler for this model. Finally, in this chapter, we also discussed the works related to this thesis.

Chapter 3

Design of Hardware Task Superscalar

The objective of this chapter is to describe the proposed designs of the hardware Task Superscalar architecture. Based on the original design, three hardware designs for Task Superscalar architecture have been proposed. The operational flow of the three hardware designs is explained and compared to the original design. For each design, two operational flows are described: one that shows the process to be done when a new task arrives, and another that shows the process to be done once a task finishes.

3.1 Operational Flow of Hardware Task Superscalar

In this section, the operational flow of each proposed hardware Task Superscalar Architecture design is described.

The operational flow of the original Task Superscalar architecture [6] was modified to design the first Hardware Task Superscalar called HTSS.1. Most of the modifications to the initial design aim at solving some stalls as well as simplifying time-consuming operations. Then, HTSS.1 was modified in order to reduce the packet communication and the cycles for processing those packets. That was done by joining two of the modules of HTSS.1. The new design (HTSS.2) was also improved by reducing the latency of entering a new task into the hardware system. This last modified version was called HTSS.3.

Finally, the new designs are compared to the initial design of the Task Superscalar.

3.1.1 Operational Flow of HTSS.1

The operational flow of the base hardware Task Superscalar architecture is shown in Figure 3.1. Figure 3.1-a shows the general operational flow to process a new task that arrives to the front-end and Figure 3.1-b shows the general operational flow to process a finished task.

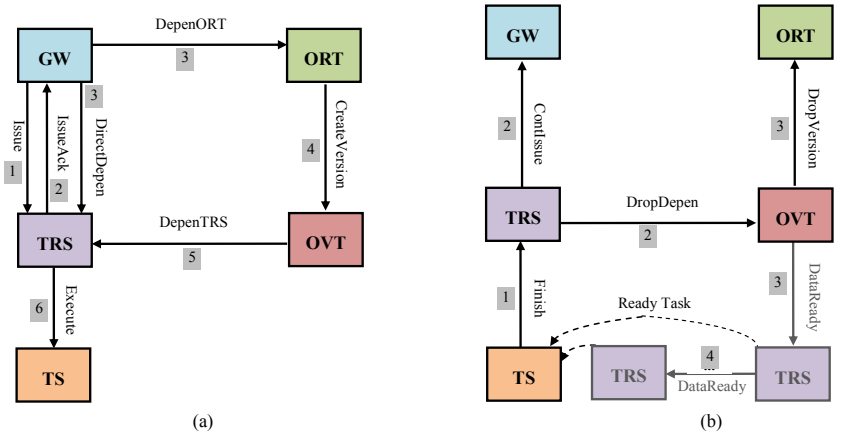


Figure 3.1: Operational flow of hardware Task Superscalar, (a) when a task arrives to the pipeline, (b) when a task is finished

Table 3.1: Definition of the packets

Packet	Description	Source/Destination
ContIssue	notifies the GW that an space in the TRS memory is available.	TRS/GW
CreateVersion	creates and/or updates an OVT entry.	ORT/OVT
DataReady	notifies another task(s) that a dependence is ready.	TRS or OVT/TRS
DepeneORT	includes a non-scalar dependence for data dependency analysis.	GW/eORT
DepenORT	includes a non-scalar dependence for data dependency analysis.	GW/ORT
DepenTRS	sends a decoded dependence.	OVT/TRS
DirectDepen	sends information of a direct (scalar) dependence.	GW/TRS
DropDepen	informs the releasing of a dependence.	TRS/OVT
DropVersion	gets permission for releasing a version.	OVT/ORT
Execute	includes the meta-data of a ready task for executing.	TRS/TS
Finish	notifies TRSs that execution of a task has been finished.	TS/TRS
Issue	includes meta-data of a task.	GW/TRS
IssueAck	is acknowledges task allocation.	TRS/GW
NextAdrs	includes a free address of TRS memory.	TRS/GW

We use Figures 3.1-a and 3.1-b to describe the general operational flow between the front-end modules. 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 C.1 describes the packets that the modules use to communicate to each other. The packets are described with more detail in Appendix C.

In addition, we also present a more detailed description of the operational flow of each of the modules described in the general view. Figures 3.2, 3.3, 3.4, and 3.5 illustrate the detail for the operational flow of GW, TRS, ORT and OVT modules, respectively. All modules (but the GW), need to wait for a packet in one of their input FIFOs before they can proceed to perform the corresponding operations. The GW needs to wait for new tasks. We will refer to those figures along the description of the general operational flow.

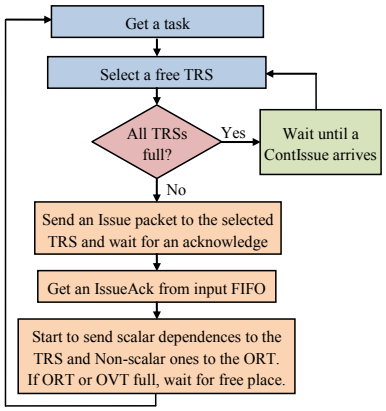


Figure 3.2: Operational flow of the GW

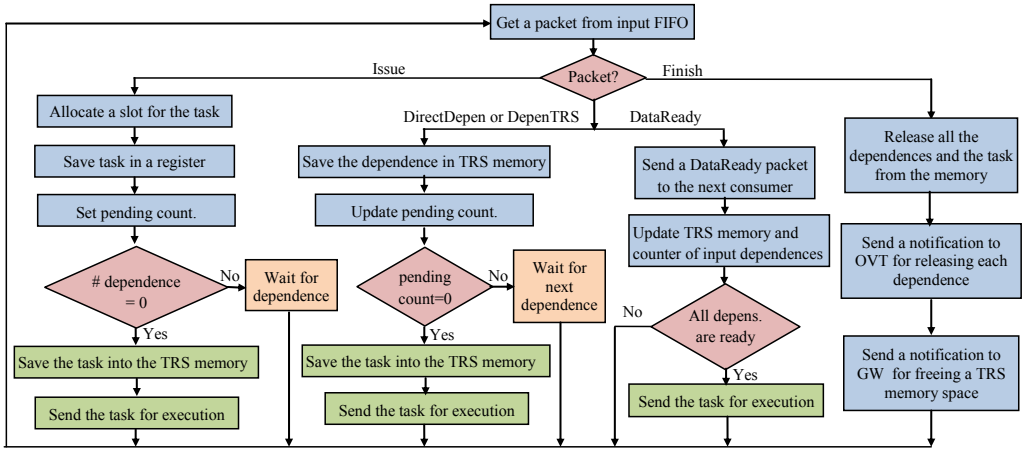


Figure 3.3: Operational flow of the TRS

Task Arrival Operational Flow

Tasks arrive to the system through the GW. As Figure 3.1-a shows, when a new task arrives, the GW sends an allocation request to one of the TRSs (`Issue` packet, sequence order 1 in Figure 3.1-a, i.e., Figure 3.1-a(1)). The detailed operational flow of the GW module is shown in Figure 3.2. First, the GW gets a task from the task generator thread. Then, it selects a TRS that has space to store the task meta-data information. If all the TRSs are full, the GW waits until a `ContIssue` packet arrives (Figure 3.1-b(2)), which means that another 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 the space for task meta-

data information. The operational flow of the TRS is shown in Figure 3.3. When a TRS gets an `Issue` packet, it allocates a slot for the task and its dependences, and sends an `IssueAck` packet, with the allocated address, to the GW (Figure 3.1-a(2)). Each task is thus represented by a unique task identifier (task ID) composed of the TRS index and the slot ID. The task ID is also used to derive unique dependence IDs, consisting of the task ID and the dependence index within the task.

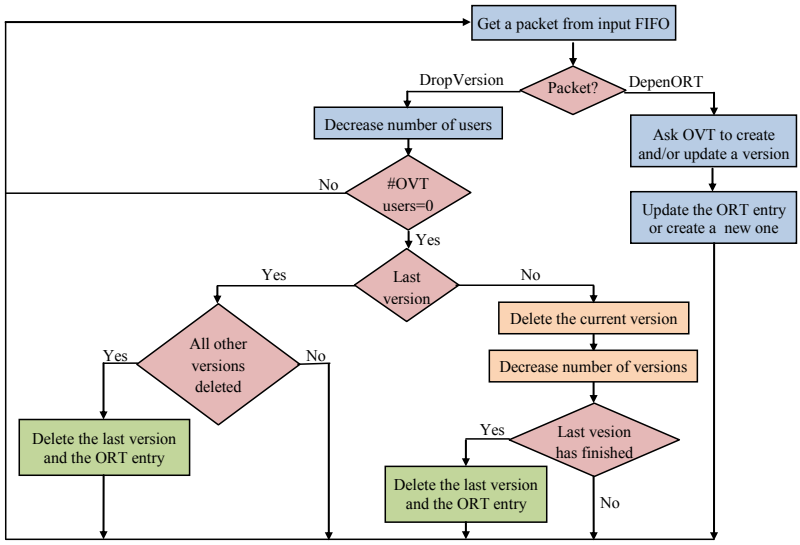


Figure 3.4: Operational flow of the ORT

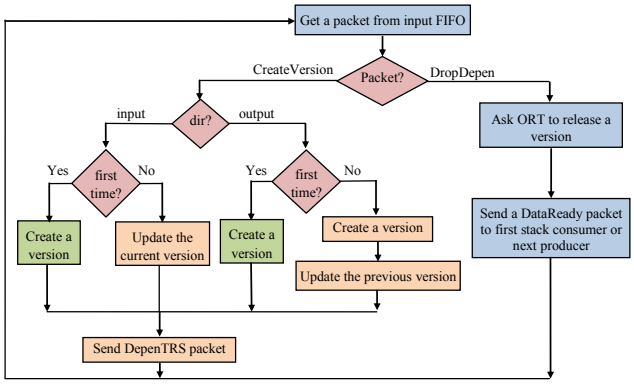


Figure 3.5: Operational flow of the OVT

Once a TRS slot is allocated, the GW starts to issue the dependences of the allocated

task to the pipeline. Scalar dependences are sent directly to the allocated TRS using `DirectDepen` packet (Figure 3.1-a(3)). For non-scalar dependences, the GW initiates the dependency decoding by sending dependences to the ORTs. To do this, the GW sends a `DepenORT` packet (Figure 3.1-a(3)) for each non-scalar dependence. If the ORT (or the OVT) does not have free space, the GW waits until they have space as described in Figure 3.2.

The functionality of the ORT (shown in Figure 3.4) is similar to the register renaming table. When `DepenORT` packet arrives to the ORT from the GW, the ORT checks if an entry for this dependence already exits, and sends a `CreateVersion` packet to activate the OVT. With this packet, the ORT asks the OVT to create a new version, or update an existing version or do both operations, depending on the direction of the dependence and whether the dependence appears for the first time or not. Meanwhile, if there is an entry for the dependence, the ORT updates it; otherwise, a new entry is created (see Figure 3.4).

As Figure 3.5 shows, when an OVT gets a `CreateVersion` packet from its ORT, it checks if it is an input or output dependence: for each output dependence, OVT creates a new version (a new producer) and updates the previous version of the dependence if it exists. For each input dependence, if it is the first time that the dependence appears, the OVT creates a new entry for it; otherwise, the OVT updates the last version of the dependence adding a new consumer. Meanwhile, the OVT sends a `DepenTRS` packet to the TRS, which includes the information of the dependence, its previous user and its related version (Figure 3.1-a(5)). Using this information, the producer/consumer chain is created in the TRSs and the OVTs.

When a TRS gets a `DepenTRS` packet, it updates the corresponding task information. However, receiving a `DepenTRS` packet for every dependence does not mean that they are ready. Usually, the task will keep waiting for `DataReady` packet (or a set of packets as explained later) until it is ready to execute. When all the input dependences of a task are available, that task becomes ready and it will be sent to the TS using an `Execute` packet (Figure 3.1-a(6)). Then, the TS sends the task to the execution units.

Since this design does not use any renaming method (for consumers or producers), we do not need to create a copy of the dependence, so its original address is used. It is essential to note that without renaming, bidirectional (i.e., inout) dependences are processed equal to output dependences.

Task Ending Operational Flow

Figure 3.1-b shows the procedure when a task is finished. When a task is finished, the corresponding TRS gets a `Finish` packet from the TS. Then, that TRS starts to release the dependences of the task. For each of the dependences, the TRS sends a `DropDepen` packet to the OVT. After all the dependences of the task are released, the TRS frees the task memory (TM) entry of the task and, if the memory was full, sends a `ContIssue` packet to the GW indicating that a slot has become free and there is free space for allocating new tasks (Figure 3.1.b(2)).

At the same time, when the OVT gets a `DropDepen` packet, it sends a `DropVersion` packet to the ORT (see Figure 3.1.b(3)). Meanwhile, for each output dependence, if there are consumers waiting, the OVT notifies the TRS that is on the top of the consumer stack that the dependence is ready by sending a `DataReady` packet (Figure 3.1-b(3)). When a consumer TRS gets a `DataReady` packet for a dependence, it updates the associated memory entry and, if there is another TRS consumer, it passes the `DataReady` packet to that TRS (Figure 3.1.b(4)). In this way, the `DataReady` packet is propagated through a producer and its consumers based on a consumer chain until there are no more consumers in the stack. When each of the consumers finishes, it sends a `DropDepen` packet to the OVT (Figure 3.1.b(2)). The OVT collects them and, after getting all of them (i.e., the version of the dependence is no longer used), it sends a `DataReady` packet to the next producer of the dependence, if it exists. This message notifies the next producer that it can be executed.

A graph transformation referred to as consumer chaining [6] is used. The consumer chaining eliminates one degree of freedom. The transformation, illustrated in Figure 3.6, chains the consumers as a linked list. This requires storing only the operand ID of the first data consumer, instead of a per operand consumer list. The chaining effectively blurs the roles of producer and consumer, as each consumer serves as its successor's producer. In Figure 3.6, for example, task C1 is a consumer from task P's point of view, but a producer for task C2. When a real producer task finishes, it sends a message to the first consumer in the chain, which immediately forwards the message to the next consumer. Therefore, in this case all the consumers of a producer are notified about the readiness of a produced dependence. Although chaining induces increasingly longer message latencies, it does not have considerable impact on the performance [6] of

Algorithm 1: HTSS.1 algorithm for processing a new task

```
1  The GW gets meta-data of a task and its dependences;
2  The GW selects a free TRS based on a round robin algorithm;
3  The GW sends the task to the allocated TRS;
4  The allocated TRS gets the issued task and sends an Ack. to the GW;
5  When the GW gets the Ack., starts to send the dependences of the task;
6  if #dependences = 0 then
7  |  TRS sends the task for execution;
8  else
9  |  for each dependence of the task do
10 |  |  if dependence is an scalar then
11 |  |  |  The GW directly sends the dependence to the TRS;
12 |  |  |  The TRS saves it in the TM;
13 |  |  else
14 |  |  |  The GW sends each non-scalar dependence to the ORT for data dependency
   |  |  |  analysis;
15 |  |  |  The ORT saves the dependence in the DM;
16 |  |  |  if dependence is an input then
17 |  |  |  |  if first time then
18 |  |  |  |  |  The ORT ask the OVT to create a version for the dependence in the VM;
19 |  |  |  |  else
20 |  |  |  |  |  The OVT updates the current version of the dependence in the VM;
21 |  |  |  |  end
22 |  |  |  else
23 |  |  |  |  The ORT asks the OVT to create a version for the dependence in the VM;
24 |  |  |  |  if NOT first time then
25 |  |  |  |  |  The OVT updates the previous version of the dependence in the VM;
26 |  |  |  |  end
27 |  |  |  end
28 |  |  |  The OVT sends the dependence with its version-id to the TRS;
29 |  |  |  The TRS saves the dependence in the TM;
30 |  |  end
31 |  end
32 |  if all the dependences are ready then
33 |  |  The TRS sends the task for executing;
34 |  end
35 end
```

real applications, and implementing this approach is much simpler than implementing broadcast of a ready message between all of its consumers.

When the ORT receives a `DropVersion` packet (Figure 3.1.b(3)), it decreases the total number of users of the dependence. If there are no more users for the version it may be deleted (details of the ORT operational flow are shown in Figure 3.4). In the case that the version is the last and there are no more users for the dependence, the ORT entry is deleted and the version in the version memory (VM) is freed. In the case that the version is not the last one and there is no more users for the dependence, the

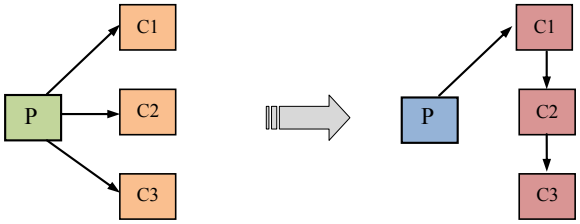


Figure 3.6: Producer/consumer chaining

ORT frees two entries of the OVT list (the version that is not going to be used more and the last version) and also deletes the corresponding ORT entry. This behavior is due to the fact that the last version of a dependence should not be deleted until all previous (other) versions are deleted. The last version should be retained because if a new consumer arrives, it should be linked to the last version and not to the one in use. Therefore, as Figure 3.4 shows, as a response to a `DropVersion` Packet, in some cases ORT asks the OVT to free one entry of its memory, in some cases, it asks to free two entries, and in some cases it does not free any entry of the version memory.

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

Algorithms 1 and 2 summarize the operational flow of HTSS.1 when it processes a new arrived task and a finished task.

Algorithm 2: HTSS.1 algorithm for processing a finished task

```
1  TRS releases all dependences and then the task from its memory;
2  The TRS notifies the OVT for each dependence;
3  if dependence is an output then
4  |  The OVT notifies readiness of the dependence to the TRS which is the top element of the
   |  consumer stack;
5  |  Last consumer of the processed version will notify next producer (next version), if exists;
6  end
7  if #users of the version = 0 then
8  |  if the version is the last one then
9  |  |  if all other version deleted then
10 |  |  |  The ORT deletes the last version and the ORT entry;
11 |  |  end
12 |  else
13 |  |  The ORT deletes the version;
14 |  |  if all other version deleted then
15 |  |  |  The ORT deletes the last version and the ORT entry;
16 |  |  end
17 |  end
18 end
```

3.1.2 Case Study

In this Section, the operational flow of the Hardware Task Superscalar is shown through two simple examples. In those examples, one GW, one ORT, one OVT and two TRSs in HTSS.1 are assumed.

Figures 3.7 and 3.8 show examples of non-dependent and dependent tasks, respectively. In these examples, there are five instances of task T (`T1, T2, T3, T4, T5`), and each of them has two defined dependences. 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 TRS1 and `T2` and `T4` are stored in TRS2.

Figure 3.7-a shows the OmpSs code of the non-dependent tasks example. Function `T` is a task with two dependences: `p1` is an output dependence and `p2` is an input dependence. The `omp` pragma is used to indicate a task and its dependences to the compiler. As all of these tasks are non-dependent, they can be executed in parallel as fast as their defined dependences are processed. Figure 3.7-b shows the allocated slots and versions of these five non-dependent tasks. Since there is no dependency between tasks of this example, there is not any producer-consumer chain here. `Vxi` is the only version of the dependence `xi` that is stored in the VM.

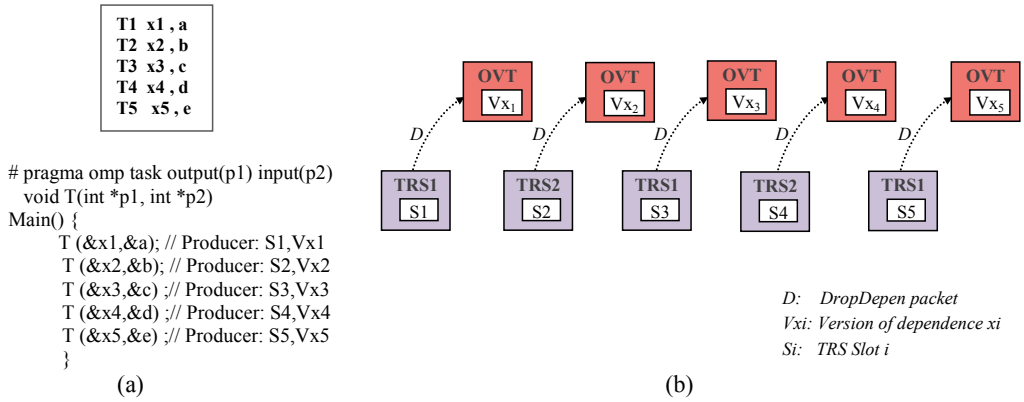


Figure 3.7: Example of five non-dependent tasks, (a) OmpSs program, (b) Relationship between TRS entries and OVT entries

Figure 3.8 presents an OmpSs program with a set of five dependent tasks through a producer-consumer chain of the variable `x`. `Vix` is a version of `x` in the VM. Task `T1` is the first producer for `x` and `T2` and `T3` are its consumers. Therefore, tasks `T2` and `T3` depend on `T1`. `V1x` is the corresponding version for these three tasks that is created when task `T1` arrives to the pipeline, because `T1` is the first producer of the dependence `x`. Version `Vix` is updated when consumers `T2` and `T3` arrive to the pipeline (no new versions for `x` are created). `T4` is another producer for `x` and `T5` is its consumer. For these two tasks, we have version `V2x` in the VM. When `T4` arrives to the pipeline, a new version (i.e., `V2x`) is created for it while the previous version, `V1x`, is updated, pointing to the current version. As `T5` is the consumer of the task `T4`, `V2x` is updated when `T5` arrives to the pipeline.

When `T1` finishes, the TRS1 which is responsible for `T1` and have saved `S1` sends a message (called *DropDepen* packet) to the OVT in order to notify it to release version `V1x`. Then, the OVT sends a ready message (`R1`) to notify the readiness of `x` to the TRS that saved `S3` which is on the top of the consumer stack. The TRS of `S3` immediately forwards the ready message to the TRS that saved `S2` which is another (and also the last) consumer of the version of `x` produced by `T1`. As soon as all the consumers of a producer finish, the next producer can execute. In this example, whenever `T2` and `T3` finish, their TRSs inform the OVT that stores `V1x`. Then, the OVT that stores `V1x` sends a ready message to the TRS2 which saves the next producer of `x` (i.e., `T4`) and

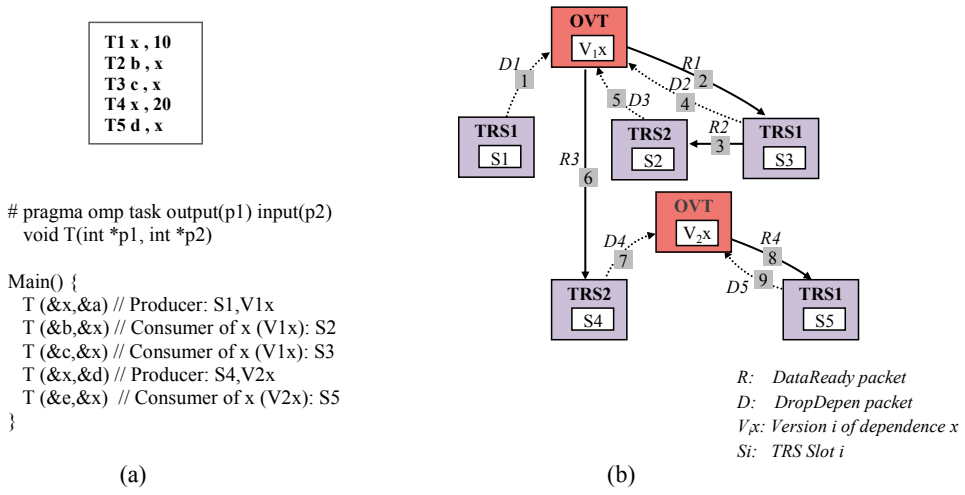


Figure 3.8: Example of five dependent tasks, (a) OmpSs program, (b) Relationship between TRS entries and OVT entries (producer-consumer chain)

deletes the memory entry of `V1x`. A similar scenario happens for `T4`, `V2x` and `T5`.

3.1.3 Operational Flow of HTSS.2

In order to improve the design of the hardware Task superscalar, the functionalities of the ORT and the OVT have been merged into a new module called extended ORT (eORT) as can be seen in Figure 3.9. The reason for this improvement is removing the redundant data transfers between these two modules. We will show in Sections 4.3.2 and 6.3 that applying this improvement reduces latency, traffic and hardware resources.

Using the new module (i.e., eORT), an improved design of the hardware Task Superscalar called HTSS.2 is presented. Algorithm 3 shows the operational flow of HTSS.2 when processing new tasks, and Algorithm 4 shows the operational flow when processing finished tasks. As the algorithms show, the eORT performs the operations of both the ORT and the OVT, it manages the dependences and versions, and maintains both previously presented memories. For the sake of brevity from now on the old ORT memory will be called Dependence Memory (DM) and the old OVT memory will be called Version Memory (VM). Consequently the TRS memory will be referred as Task Memory (TM).

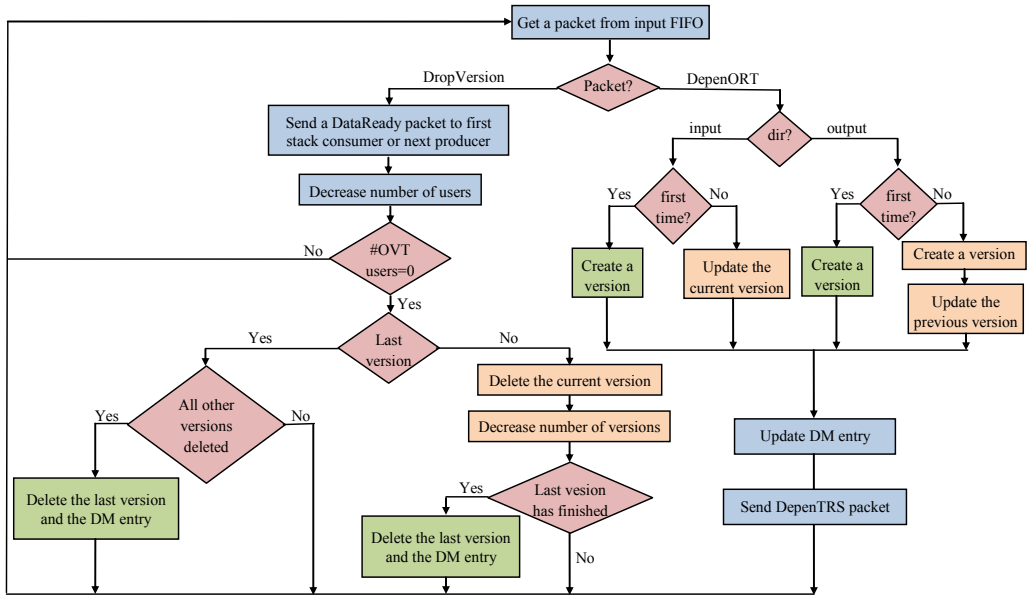


Figure 3.9: Operational flow of the eORT module

Although it can be argued that some parallelism is lost by joining the two modules, in this case, experiments have shown that saving the work of managing the packets between them overcomes the disadvantages. However, as will be shown in next chapters, the same does not apply to TRS and eORT modules.

3.1.4 Operational Flow of HTSS.3

In this section the third version of hardware Task Superscalar architecture which is called HTSS.3 is described. For this version, an improvement to HTSS.2 design is applied in order to reduce the latency cycles required to process a new task. The improvement is applied to the procedure of issuing a task and its dependences to the pipeline. Figures 3.10 and 3.11 show the operation of the new GW (called improved GW — iGW) and the new TRS (called improved TRS — iTRS), respectively. Those figures are explained through their new workflow algorithms.

Algorithms 5 and 6 describe the operational flow of HTSS.3 when a new task arrives and when a task finishes, respectively. In this design, the iGW has a new internal register that stores the address of a free slot in the task memory of each available iTRS

Algorithm 3: HTSS.2 algorithm for processing a new task

```
1  The GW gets meta-data of a task and its dependences;
2  The GW selects a free TRS based on round robin algorithm;
3  The GW sends the task to the allocated TRS;
4  The allocated TRS gets the issuing task and send an Ack. to the GW;
5  When the GW gets the Ack., starts to send the dependences of the task;
6  if #dependences = 0 then
7  |  The TRS sends the task to execute;
8  else
9  |  for each of the dependences do
10 |  |  if dependence is an scalar then
11 |  |  |  The GW directly sends the dependence to the TRS;
12 |  |  |  The TRS saves it in the TM;
13 |  |  else
14 |  |  |  The GW sends each non-scalar dependence to the eORT for data dependency
   |  |  |  analysis;
15 |  |  |  eORT saves the dependence in the DM;
16 |  |  |  if dependence is an input then
17 |  |  |  |  if first time then
18 |  |  |  |  |  The eORT creates a version for the dependences in the VM;
19 |  |  |  |  else
20 |  |  |  |  |  The eORT updates the current version of the dependence in the VM;
21 |  |  |  |  end
22 |  |  |  else
23 |  |  |  |  The eORT creates a version for the dependence in the VM;
24 |  |  |  |  if NOT first time then
25 |  |  |  |  |  The eORT updates the previous version of the dependence in the VM;
26 |  |  |  |  end
27 |  |  |  end
28 |  |  |  The eORT sends the dependence to the TRS;
29 |  |  |  The TRS saves it in the TM;
30 |  |  end
31 |  end
32 |  if all the dependences are ready then
33 |  |  The TRS sends the task to execute;
34 |  end
35 end
```

module. Then, the iGW sends the meta-data of the task and the allocated address to the iTRS (see Algorithm 5). In this design, right after sending task meta-data, the iGW starts to issue the dependences, adding the iTRS identifier and slot address to the dependences data. When all of the dependences of a task have been sent, the iGW is ready to send an allocation request of the next task. On the other hand, when the iTRS gets a message including task meta-data, saves it in the allocated slot, and if there is more free space in the TRS, it sends a message including the new slot address to the iGW. With this mechanism the address is already in the iGW when it reads a new task

Algorithm 4: HTSS.2 algorithm for processing a finished task

```
1  The TRS releases all dependences and task from the memory;
2  The TRS notifies a DropDepen packet to eORT for each dependence;
3  if dependence is an output then
4  |  The eORT notifies readiness of the dependence to the TRS which is the top element of the
   |  consumer stack;
5  |  Last consumer of the processed version will notify next producer (next version), if exists;
6  end
7  if #users of the version = 0 then
8  |  if the version is the last one then
9  |  |  if all other version deleted then
10 |  |  |  The eORT deletes the last version and the eORT entry;
11 |  |  end
12 |  else
13 |  |  The eORT deletes the version;
14 |  |  if all other version deleted then
15 |  |  |  The eORT deletes the last version and the eORT entry;
16 |  |  end
17 |  end
18 end
```

that should be allocated in the same iTRS module.

As we explained in previous sections, in other Task Superscalar designs [6, 8, 145] (i.e., the initial design, HTSS.1 and HTSS.2), when the GW sends a task meta-data to the TRS, it has to wait for the acknowledge from the allocated TRS in order to get the TM slot address and add it to each dependence. After that, it can start to issue the dependences of a task to the pipeline. Waiting for an acknowledge message including the address where the task was allocated is an extra overhead in the procedure of issuing a task and its dependences to the pipeline. As explained, in the third design, this waiting time has been removed by saving (prefeching) a free address of the iTRS memory in a register in the iGW. The iGW allocates this address to the new incoming task and sends the task to the selected iTRS. Immediately after that, it starts to send the dependences with the allocated address. The register is updated with a new free slot address by the next ContIssue packet. This ContIssue is produced and sent by the iTRS when it gets a new task and still has free slots, or when a task is finished and there were no previously free slots (in this case the sent address is the one freed by the finished task).

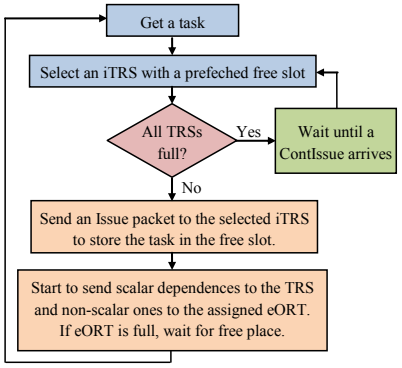


Figure 3.10: The operational flow of the improved GW (iGW) module

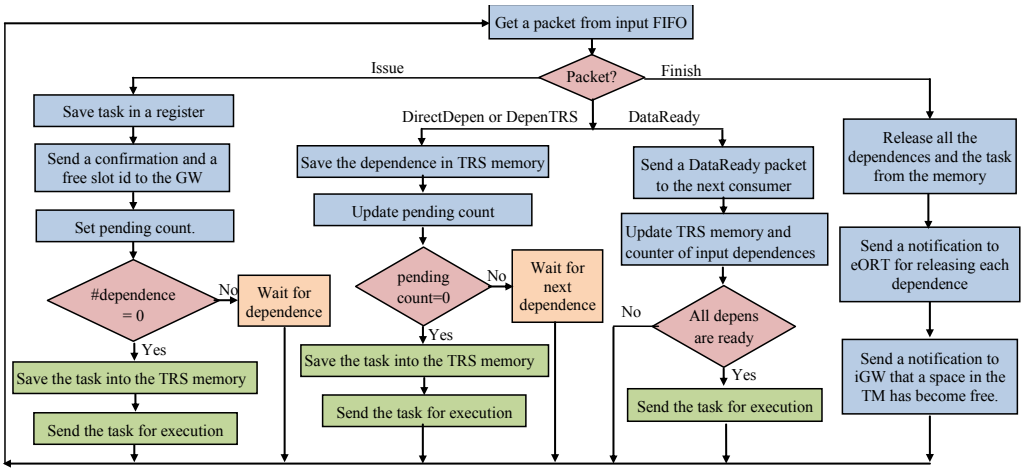


Figure 3.11: The operational flow of the improved TRS (iTRS) module

3.2 Comparison of HTSS and its predecessor

The operational flow of the hardware design of the Task Superscalar is different in several aspects from the original version that was presented by Etsion et al. [6, 7] (called TSS). Here, the differences are summarized.

The operational flow of processing arrived and finished tasks of the original Task Superscalar has been modified. Therefore, the hardware designs have fewer packets that are also denser than the packets used in the initial design. One of the modifications is the procedure for sending non-scalar dependences in order to avoid possible stalls. In addition, this modification also removes two packets that were used in the original ver-

Algorithm 5: HTSS.3 algorithm for processing a new task

```
1  The iGW gets meta-data of a task and its dependences;
2  The iGW selects a free iTRS based on round robin algorithm and free slot availability;
3  The iGW sends the task with a slot address to the allocated iTRS;
4  The iGW starts to send the dependences of the task;
5  if #dependences = 0 then
6  |  The iTRS sends the task to execute;
7  else
8  |  for each of the dependences do
9  |  |  if dependence is an scalar then
10 |  |  |  The iGW directly sends the dependence to the iTRS;
11 |  |  |  The iTRS saves it in the TM;
12 |  |  else
13 |  |  |  The iGW sends each non-scalar dependence to the eORT for data dependency
   |  |  |  analysis;
14 |  |  |  The eORT saves the dependence in the DM;
15 |  |  |  if dependence is an input then
16 |  |  |  |  if first time then
17 |  |  |  |  |  The eORT creates a version for the dependences in the VM;
18 |  |  |  |  else
19 |  |  |  |  |  The eORT updates the current version of the dependence in the VM;
20 |  |  |  |  end
21 |  |  |  else
22 |  |  |  |  The eORT creates a version for the dependence in the VM;
23 |  |  |  |  if NOT first time then
24 |  |  |  |  |  The eORT updates the previous version of the dependence in the VM;
25 |  |  |  |  end
26 |  |  |  end
27 |  |  |  The eORT sends the dependence to the iTRS;
28 |  |  |  The iTRS saves it in the TM;
29 |  |  end
30 |  end
31 |  if all the dependences are ready then
32 |  |  The iTRS sends the task to execute;
33 |  end
34 end
```

sion. Figure 3.12 shows the difference. In the original version, non-scalar dependences were sent to the ORT. After that, the ORT sent a request for creating a version to the OVT and after that, the ORT passed the dependence to the TRS. Then, the TRS asked the OVT for the address of the version in the OVT memory (i.e., VM). After processing the request for creating a version, the OVT informed the TRS the address of the version. In contrast, in our design, the OVT is responsible for sending both the dependence and address of the version to the TRS, after processing the request for creating a version. With this modification the functionality is maintained diminishing the time needed to process a task. In addition, a deadlock caused by the TRS waiting for the address of a

Algorithm 6: HTSS.3 algorithm for processing a finished task

```
1  The iTRS releases all dependences and then the task from its memory;
2  The iTRS notifies a DropDepen packet to eORT for each dependence;
3  if dependence is output then
4  |  The eORT notifies readiness of the dependence to the iTRS which is the top element of the
   |  consumer stack;
5  |  Last consumer of the processed version will notify next producer (next version);
6  end
7  if #users of the version = 0 then
8  |  if the version is the last one then
9  |  |  if all other version deleted then
10 |  |  |  The eORT deletes the last version and the eORT entry;
11 |  |  end
12 |  else
13 |  |  The eORT deletes the version;
14 |  |  if all other version deleted then
15 |  |  |  The eORT deletes the last version and the eORT entry;
16 |  |  end
17 |  end
18 end
19 TRS sends a message to the iGW to inform it the freed slot address;
```

dependence in an OVT that has run out of memory is avoided.

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

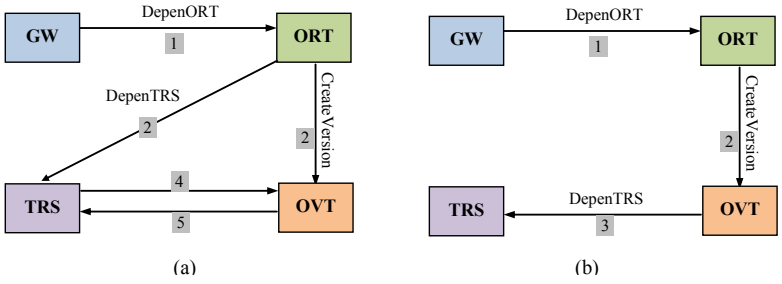


Figure 3.12: Operational flow of sending non-scalar dependences, (a) the original version, (b) the hardware version

3.3 Summary

This chapter has presented three hardware Task Superscalar designs: A base working design that preserved the original functionality of the Task Superscalar architecture and can be implemented in VHDL, and two improved designs. The differences between the hardware designs have been explained along with the necessary modifications to the original proposal.

Chapter 4

Hardware Prototypes for Latency Exploration

This chapter explores the latencies of the three proposed HTSS designs. For this, three prototypes of the HTSS are presented following the three designs that have been proposed in the previous chapter. The prototypes have been written in VHDL in order to simulate them with an HDL simulator at register transfer level (RTL). As a result of this exploration, the required latencies for creating, processing, storing and retrieving all the packets in the different HTSS prototypes have been obtained. These packets are used for transferring information of tasks and their dependences between the different components of the HTSS. With these results a design space exploration of HTSS is performed in the next chapter. In this chapter, first, the three hardware prototypes are described as implementations of the three proposed designs and the challenges of hardware prototyping the Task Superscalar are discussed. Second, the methodology and experimental setup of the hardware prototyping are described. Finally, the obtained results from the latency exploration are presented.

4.1 Hardware Prototypes

In this section, the hardware designs of the Task Superscalar prototypes are described. The purpose for this prototyping is to determine and evaluate the required latencies for processing all the packets in the system under various circumstances. Those prototypes are based on the operational flows explained in Section 3.1. Figure 4.1 shows the high level description of HTSS. In this figure, it is shown that HTSS is connected to a trace generator thread and several processors that operate as workers for executing decoded tasks.

Although Etsion et. al [6] determined that a Task Superscalar configuration with eight TRSs, two ORTs and two OVTs was enough to support a 256 processor system, for simplicity, the prototypes have been designed with one GW, two TRSs, one ORT/OVT (or one eORT) and one TS. It is important to mention that the number of components have been determined only to have hardware-described prototypes of the different versions of the Task Superscalar architecture in order to perform a detailed analysis of each component. In the following chapter, these latencies are used to perform a full design space exploration of the optimal number of components and memory capacities of HTSS in a real implementation.

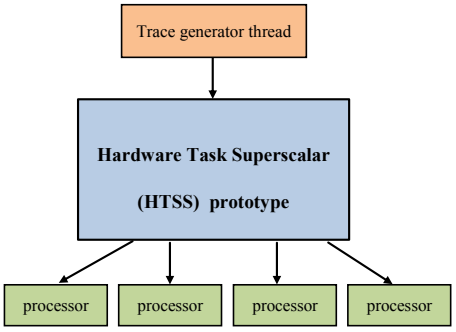


Figure 4.1: Hardware Task Superscalar architecture

4.1.1 HTSS.1 Prototype

HTSS.1 Task Superscalar architecture prototype has been implemented based on the operational flow that has been described in Section 3.1.1. Figure 4.2 illustrates the high-level diagram of the implementation of the design. This implementation is an adaptation of the high level description of the pipeline of the Task Superscalar into

VHDL code. The main modules of HTSS.1 communicate to each other using messages (packets). The prototype is mainly composed of one GW, two TRSs, one ORT, one OVT, one TS, 23 FIFOs and 6 arbiters.

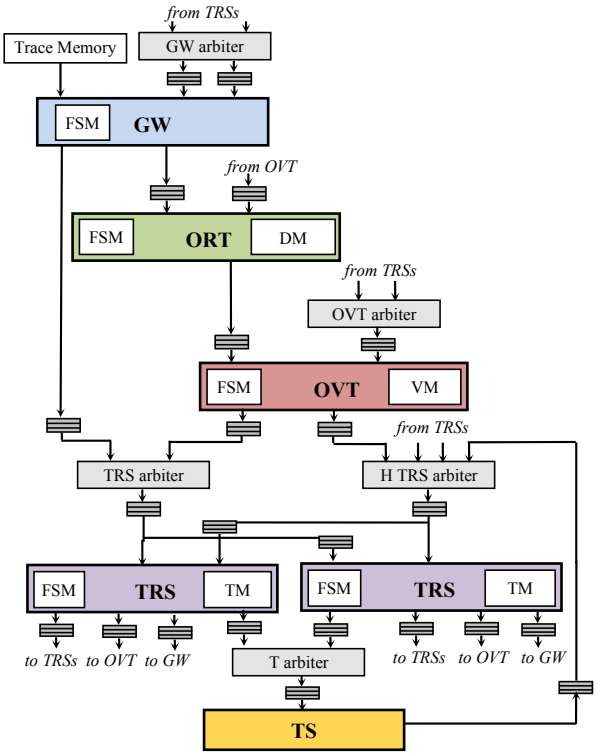


Figure 4.2: HTSS.1 prototype

One main consideration in the hardware prototype implementation has been reducing the FPGA resources used in controllers, buses, and registers to save FPGA resources for the memory units of the modules.

As explained, in this design there are three kinds of memory modules: Task memory (TM), dependence memory (DM) and version memory (VM). The detailed characteristics of those memories are explained in Appendix B. Figure 4.3 shows the structure of those memories. Using the original TSS results, the capacities of the TM and the DM have been assumed to be equal to 200 Kbytes while the VM has been designed to use 160 Kbytes. In the hardware prototype, the TM is divided into slots. Each slot has sixteen 200-bit entries, one for every meta-data of the task (task description entries and 15 dependence information entries). With this size, each TRS can store up to 512

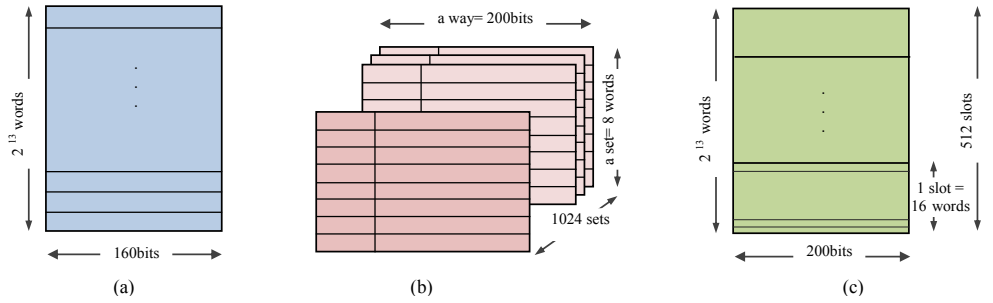


Figure 4.3: Structure of the memory modules: (a)VM, (b)DM, (c)TM

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 VM can save up to 8K versions of dependences. The VM and the TM are direct mapped memories. The DM is an 8-way set-associative memory with 1024 sets for storing the meta-data of 8K dependences. Associativity is necessary in the case of DM to be able to detect dependences using the memory address. Otherwise, a sequential search would be necessary. Table 4.1 summarizes the characteristics of the memories. For each kind of memory, its capacity, kind of associativity, size and number of entries, size of address bus (*adrs*), data bus (*data*), size and number of slots, size and number of way, tag and index width are listed. A RAM, called trace memory, has been used to save the generated traces in order to test the design. Each task includes 17 80-bit entries, two entries for saving the meta-data of the task and the others for saving up to 15 dependences.

Table 4.1: Characteristics of the memory modules

Component	Capacity (KBytes)	Associativity	Entry size (bits)	#Entries	Details
Task Memory (TM)	200	Direct mapped	200	8K	adrs =13b, data =200b slot =16b, #slot =512
Version Memory (VM)	160	Direct mapped	160	8K	adrs =13b, data =160b

Dependency Memory (DM)	200	8-way set associative	146
			(+ 54 for tag)
				8K	#way=8, way =200b, adrs =64b,
					data =146b, index =10b, tag =54b

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 FIFOs that decouple the processing of every component

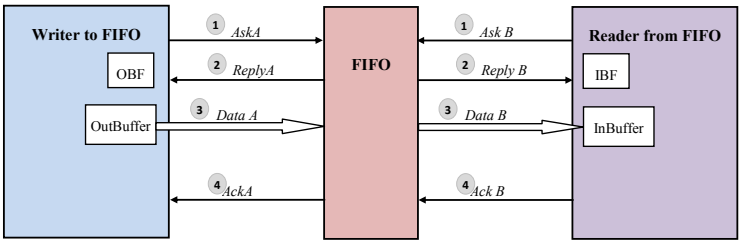


Figure 4.4: Connections of a FIFO based on 4-step handshaking protocol

in the system. This kind of network configuration allows the system to scale easily while preventing the stalls that may happen during processing a sequence of tasks and their dependences. The interconnection between the HTSS modules and FIFOs use a 4-step handshaking protocol as shown in Figure 4.4. The FIFOs are divided, mainly, in high and low priority FIFOs. The packets that free resources on the system (e.g., `Finish` or `DropDepen` packets) go to the high priority FIFOs while the packets that allocate resources go to the low priority FIFOs.

In order to minimize the width of the FIFOs and also the bandwidth of intra-module buses, packets are defined with the minimum possible size multiple of 8 bits as specified in Table 4.2. 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 uses one packet. The only corner case that makes an exception to this rule is the `Execute` message. The size of `Execute` message is related to the number of dependences and the maximum size of this packet could be up to 1295 bits; so, the data of this message is sent in sequential cycles (packets) to the TS.

Another important implementation consideration has been minimizing the cycles required for processing the system packets in order to increase the overall system throughput. The main functionality of each component is done by a finite state machine (FSM). The FSMs are designed to have a minimum number of states, each state taking only one cycle to be completed. For accessing the memories, one state initializes the control signals of the memory and the following state performs the memory accessing. When the FSM is waiting for a packet or due to system stalls, it returns to the same state until the waiting ends.

The TRS FSM processes different packets including the information of the tasks (`Issue` packet), their dependences (`DirectDepen` and `DepenTRS` packets), the notifica-

tions of termination of execution of tasks (`Finish` packet) and the readiness of a dependence (input or output) due to finished tasks that free their dependences (`DataReady` packet). Since processing `Finish` and `DataReady` packets has higher priority than processing other packets, TRSs get these packets from a separate FIFO which has higher priority than the other input FIFOs of the TRSs. The ORT FSM controls the creation and deletion of versions of dependences 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 `DepenORT` packets. The OVT FSM is responsible for creating and updating the dependences based on the decisions of the ORT. Then, the OVT sends the dependences information to the associated TRS. The OVT has also two input FIFOs: one high priority FIFO for `DropDepen` packets and the other, with normal priority, for `CreateVersion` packets.

Each module has an input signal for enabling it. When this signal is not active, all the output ports of the module remain in high-impedance. Additionally, each module has a signal to reset all output ports, intermediate signals and registers, and also to invalidate all memory words. All the modules are synchronous with a common clock signal which is rising-edge triggered.

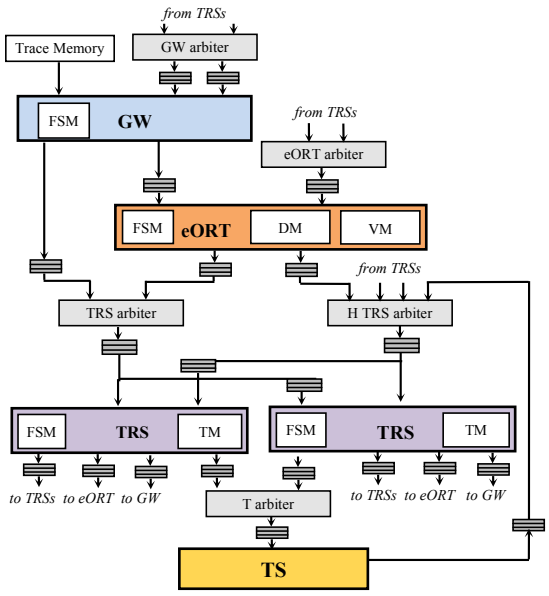


Figure 4.5: HTSS.2 prototype

4.1.2 HTSS.2 Prototype

Figure 4.5 shows the HTSS.2 prototype that is designed based on the operational flow described in Section 3.1.4. In this design, the ORT and the OVT have been merged into a new module called *extended ORT* (eORT). As Figure 4.5 shows, HTSS.2 prototype is composed of one GW, two TRSs, one eORT, one TS, 21 FIFOs and 5 arbiters.

In this design, the eORT has two memories: the DM for saving the dependences and the VM for saving the versions of each dependence. By merging the ORT and the OVT into the eORT, two FIFOs that interconnected the ORT to the OVT and the OVT to the ORT in the previous prototype were removed. Moreover, the hardware resources of the eORT are less than the hardware resources used by both the ORT and the OVT. As a result of this modification, the latency, internal communication traffic, and also the hardware resources used are reduced. Even more important, the throughput of the system is increased as will be shown in the comparison of the prototypes.

4.1.3 HTSS.3 Prototype

Figure 4.6 shows the high-level diagram of the HTSS.3 prototype which is based on the operational flow described in Section 3.2. This prototype is composed of one iGW, two iTRSs, one eORT, one TS, 17 FIFOS and 4 arbiters.

Figure 4.7 shows the interconnection between the iTRSs and the iGW of the three prototypes. In the case of HTSS.3 design each iTRS directly connects to the iGW through one slot FIFOs accessed according to the 4-step handshaking protocol.

In this design, an iTRS only sends one type of message to the iGW. This message includes a free slot address that will be used to allocate the next task. The content of this message arrives to a new FSM of the iGW, included in the HTSS.3 design. Therefore, the new design has two FSMs in the iGW. One of them processes new tasks as in previous designs, and the other reads the 1-slot FIFO from the iTRSs, and keeps updated the internal iGW register that stores an available TM entry. With this modification, the packets `IssueAck` and `ContIssue` are joined, and the iGW knows, in advance, which iTRS still has empty space by checking the valid bit of the iGW internal registers. With this, iGW can reduce unnecessary stalls when figuring out if there is any iTRS with empty slots.

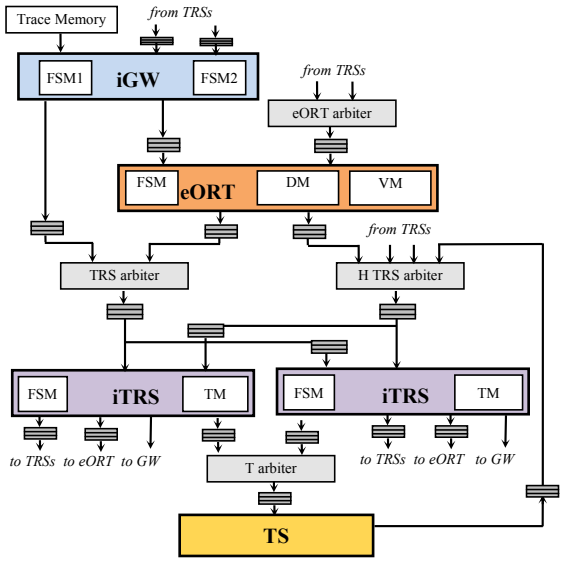


Figure 4.6: HTSS.3 prototype

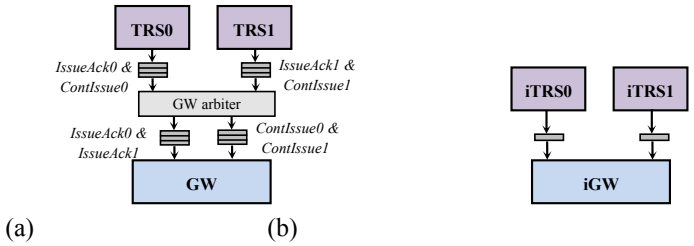


Figure 4.7: The interconnection between TRSs/iTRSs and GW/iGW (a) HTSS.1 and HTSS.2 prototypes, (b) HTSS.3 prototype

4.2 Methodology of the HDL Design of HTSS

Our hardware prototypes have been hand-coded in VHDL to be synthesized and mapped on an FPGA. Implementing a logic design with an FPGA usually consists of the following steps, depicted in Figure 4.8:

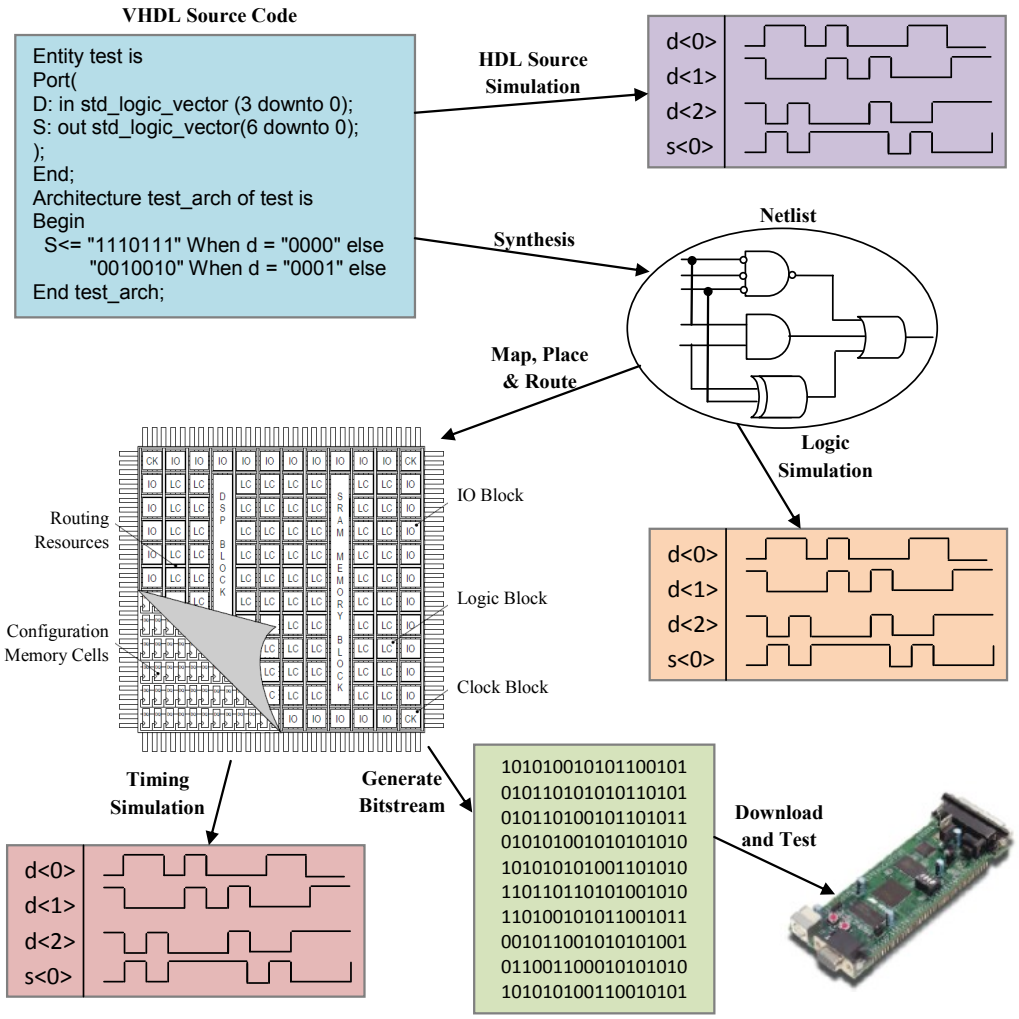


Figure 4.8: Steps of implementing a logic design with an FPGA

First, a description of a design using a hardware description language (HDL) such as VHDL or Verilog or a schematic editor is written, and tested using an HDL simu-

lator. Then, a logic synthesizer tool is used to transform the HDL or schematic into a netlist. The netlist is just a description of the various logic gates in the design and how they are interconnected. An implementation tool is used to map the logic gates and interconnections into the FPGA. The FPGA consists of many configurable logic blocks (CLB), and look-up tables (LUTs) that perform logic operations. The CLBs and LUTs are interwoven with various routing resources. The mapping tool collects the netlist gates into groups that fit into the LUTs and then the place and route tool assigns the groups to specific CLBs opening or closing the switches in the routing matrices to connect them together.

Once the implementation phase is complete, a program extracts the state of the switches in the routing matrices and generates a bitstream where the ones and zeroes correspond to open or closed switches. The bitstream is downloaded into a physical FPGA chip (usually embedded in some larger system). The electronic switches in the FPGA open or close in response to the binary bits in the bitstream. Upon completion of the downloading, the FPGA will perform the operations specified by the HDL code or schematic.

To verify the functionality of each module, the hardware design has been simulated using ModelSim 6.6d of Altera and also Isim tool of the Xilinx ISE using several bit traces. Those traces represent input packets that test the main and corner (e.g., chains of inout dependences) working conditions of the system. The correctness of the output packets generated by the modules and also the modifications to their related memories have been tested.

4.3 Latency Exploration Results

In this section, the RTL simulation results of the three hardware prototypes are presented. The objectives are: (1) to analyze the latency of processing each packet, (2) to analyze the latency of managing one isolated task with different number and types of dependences. For this, the number of cycles obtained by the HDL simulator for processing the packets and 15 different cases that represent the best, the worst and the average cases for isolated tasks are presented. After that, the influence of the data dependency management of several tasks in the system performanceis analyzed.

4.3.1 RTL Simulation Results

Table 4.2 presents the latency (in cycles) required by each FSM to process the received packets as well as the responsible unit (i.e., the unit that receives the packet) and the size of the packets. The detailed characteristics of those packets are explained in the tables of Appendix C. Each module has been tested with all possible different types of input packets using an HDL simulator (i.e., ModelSim). The latencies of accessing the memory and interconnection modules are also presented in Table 4.2.

The processing of each packet, depending on the type of packet, may have accesses to memories and produce another packet. Therefore, the processing of a packet may have different latency depending on the carried information and the internal state of the system. For example, the simulation results show that processing a `CreateVersion` packet for an output dependence that does not appear for the first time in the system takes four cycles, and three cycles in the case of an input dependence. However, when the same packet carries an input or output dependence that appears for the first time in the system, it is processed in only two cycles. Processing packets that include metadata of a task and all of its dependences takes a number of cycles that depends on the number of dependences. Among those types of packets, the processing of the `Finish` packet takes most of the time. In addition to the latencies shown in Table 4.2, each module uses four extra cycles in order to check the input packets, select the one with the highest priority and initialize its FSM.

Table 4.3 shows the latencies that the different prototypes use for processing isolated tasks as a function of the quantity of the task dependencies and their types. The latencies are obtained from the HDL simulator and counted from the moment that a

Table 4.2: Latencies of processing the packets

Packet	Processing latency	Responsible Unit	Size (bits)
ContIssue	2 cycles	GW/iGW	24
CreateVersion	- 2 cycles if the (input or output) dependence appears for the first time - 3 cycles if the dependence is input and does not appear for the first time - 4 cycles if the dependence is output and does not appear for the first time	OVT	240
DataReady	2 cycles	TRS/iTRS	64
DepeneORT	- 2 cycles if the (input or output) dependence appears for the first time - 3 cycles if the dependence is input and does not appear for the first time - 4 cycles if the dependence is output and does not appear for the first time	eORT	176
DepenORT	2 cycles	ORT	168
DropDepen	2 cycles	OVT or eORT	40
DropVersion	- 2 cycles for deleting the last version - 3 cycles for deleting the last version and the previous one	ORT or eORT	120
Execute	1 cycle (for loading meta-data of a task) + 2 cycles for loading every dependence fromTM + 1 cycle for sending the task to the TS	TRS/iTRS	170+ (#Dependences*75)
Finish	3 cycles + 2 additional cycles for loading every dependence from TM	TRS/iTRS	88
Issue	- 4 cycles for tasks without any dependence - 3 cycles for tasks with at least one dependence + 2 additional cycles for every dependence (using DepenTRS (200bits) or DirectDepen (96 bits) packets)	TRS/iTRS	160
IssueAck	2 cycles	GW	24

packet reached the GW/iGW until it is ready to be sent from a TRS/iTRS to the TS. These cases have been selected to find out the minimum and maximum number of cycles that are required for processing different kinds of tasks: tasks without dependences, with one dependence, with two dependences, and with fifteen dependences (i.e., the maximum number of dependences in those prototypes). Each dependence can be scalar or non-scalar; and non-scalar dependences can be input or output and may appear in the system for the first time or not. Note that processing *inout* dependences is the same as output dependences. As the table shows, the minimum latency is for processing scalar dependence and the maximum latency is for processing output dependences that do not appear for the first time in the pipeline.

4.3.2 Latency Comparison of HTSS Prototypes

In this section, the three prototypes are compared from the point of view of performance. The latency results of the RTL simulation of the FSMs of each component of the prototypes for processing different packets have been used to estimate the total la-

Table 4.3: Latencies of processing isolated tasks

	#dependences	Condition of the dependence(s)	Latency for processing the task (cycles)		
			HTSS.1	HTSS.2	HTSS.3
Case 1	0	-	21	21	21
Case 2	1	scalar	55	55	29
Case 3	1	non-scalar (input or output) and first time	75	65	35
Case 4	1	non-scalar, input, and not first time	76	66	36
Case 5	1	non-scalar, output, and not first time	77	67	37
Case 6	2	both scalar dependences	63	63	37
Case 7	2	1st dependence: scalar 2nd dependence: non-scalar (output or input) and first time	79	69	39
Case 8	2	1st dependence: scalar 2nd dependence: non-scalar, input and not first time	80	70	40
Case 9	2	1st dependence: scalar 2nd dependence: non-scalar, output and not first time	81	71	41
Case 10	2	1st dependence: non-scalar, input (or output) and first time 2nd dependence: non-scalar, output (or input) and first time	83	73	43
Case 11	2	both non-scalar, input and not first time	84	74	44
Case 12	2	1st dependence: non-scalar, input and not first time 2nd dependence: non-scalar, output and not first time	84	74	44
Case 13	2	both non-scalar, output and not first time	85	75	45
Case 14	15	all scalar	223	223	141
Case 15	15	all non-scalar, output and not first time	245	207	149

tency of the prototypes for processing the five dependent tasks and five non-dependent tasks already introduced in Section 3.1.2. The goal is to compare the performance of the prototypes HTSS.1, HTSS.2 and HTSS.3 to show the effectiveness of the improvements applied to the base HTSS (i.e., HTSS.1).

Table 4.4 shows an estimation of the overall time and number of cycles that the data dependency analysis of the two examples take on the hardware prototypes. Note that the execution time of the tasks is the same in all the hardware prototypes as it depends on the back-end processors. Therefore, we have decided to not include the task latency in the overall count (i.e., the tasks are instantaneously computed). The table also shows the task throughput (tasks issued per second) for the hardware prototypes. As it can be seen, different clock rates have been used taking into account the working frequencies reported by the synthesis tool for each prototype. The detailed results of the synthesis of each individual module of the three prototypes are presented in Chapter 6. Here, in Table 4.4, the frequency of the slower component in each of the three prototypes has

been selected as the prototype working frequency.

The time used by the system to process a task is critical for it to be able to cope with smaller tasks in order to get enough parallelism in big systems. The initial results show that due to the tiled structure of the Task Superscalar architecture, when processing several tasks at the same time, most of the operations are simultaneously performed.

Table 4.4: Latencies, estimated execution time and task throughput of processing five tasks on the hardware prototypes

	Latency for processing tasks (cycles)			Latency for processing tasks (µs)			Task Throughput (task/ sec)		
	HTSS.1	HTSS.2	HTSS.3	HTSS.1 (153 MHz)	HTSS.2 (146 MHz)	HTSS.3 (140 MHz)	HTSS.1	HTSS.2	HTSS.3
5 non-dependent tasks	242	230	98	1.58	1.57	0.7	3099×10^3	3260×10^3	7653×10^3
5 dependent tasks	305	298	253	2	1.9	1.8	2460×10^3	2516×10^3	2964×10^3

Figure 4.9 shows the time scheduling and sequence of processing different packets of the two examples of five tasks in the modules of the hardware Task Superscalar prototypes. The packets related to processing the first task (T1) are joined with the arrows to help to follow the sequence. The legend of the figure follows:

- `Ti`: The required number of cycles that each component needs for processing packets related to Task i (for creating and processing `Issue` packet)
- `Pij`: The required number of cycles that each component needs for processing packets related to Dependence j of Task i (for creating and processing `DirectDepen`, `DepenTRS`, `DepenORT`, `DepeneORT` and `CreateVersion` packets)
- `Ti R`: The required number of cycles for preparing a packet for Task i to be ready and sent for execution
- `Finish Ti`: The required number of cycles for processing the `Finish` packet of Task i
- `Di`: The required number of cycles for processing a `DropDepen` packet (The index is the order in which this packet is created in Figures 3.7 and 3.8)

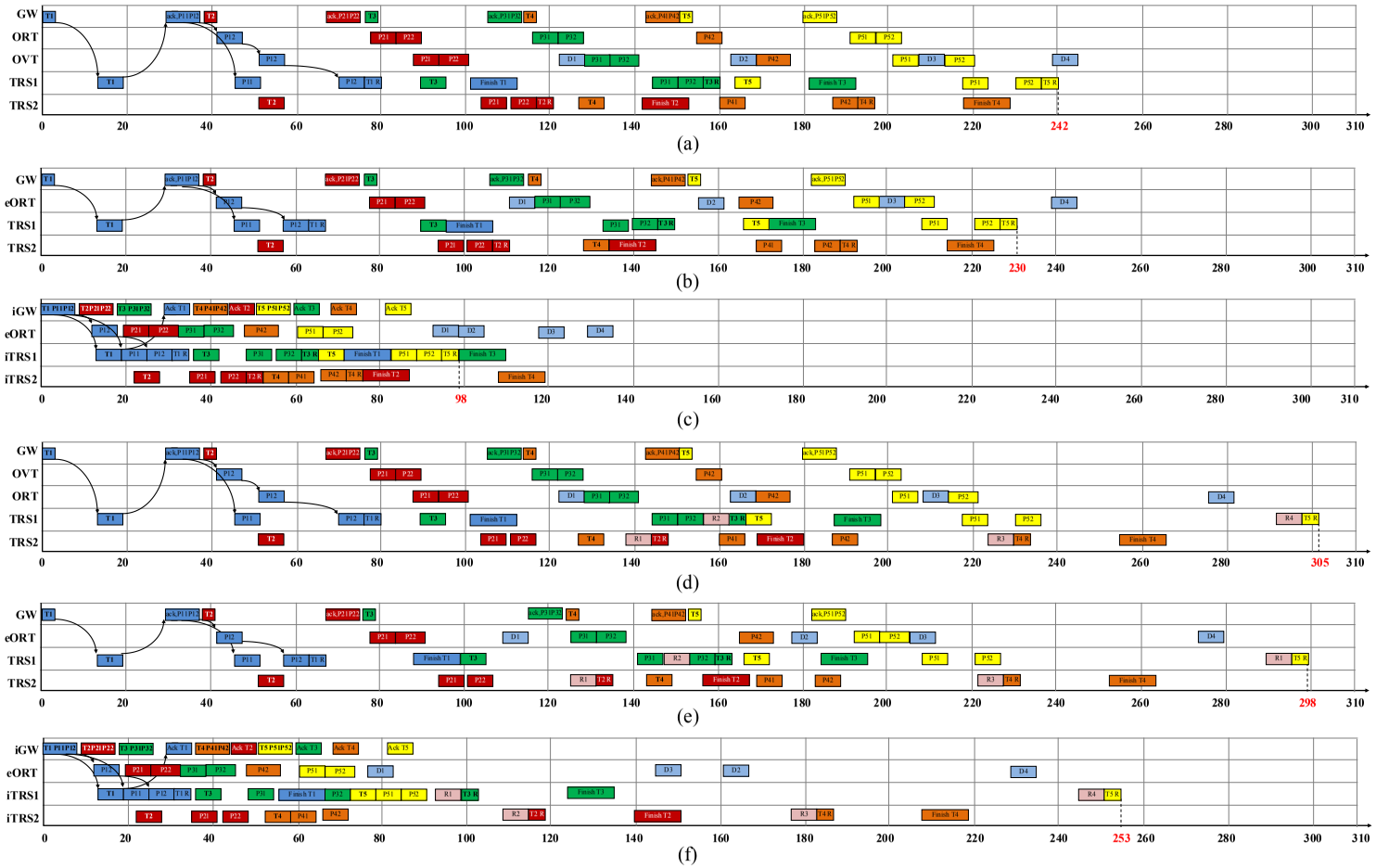


Figure 4.9: Time scheduling, in cycles, of a) 5 non-dependent tasks on HTSS.1, b) 5 non-dependent tasks on HTSS.2, c) 5 non-dependent tasks on HTSS.3, d) 5 dependent tasks on HTSS.1, e) 5 dependent tasks on HTSS.2, f) 5 dependent tasks on HTSS.3

- `Ri`: The required number of cycles for processing a `Ready` packet (The index is the order in which this packet is created in Figures 3.7 and 3.8)

In particular, Figures 4.9-a, 4.9-b and 4.9-c show the behaviour of the prototypes to process the five non-dependent tasks, and Figures 4.9-d, 4.9-e and 4.9-f for the case of five dependent tasks. Note that the `Ready` and `DropDepen` packets are related only to the output dependences.

Figure 4.9 shows that due to the tiled structure of the Task Superscalar architecture, most of the operations of processing a new task and its dependences or a finished task are accomplished simultaneously. Moreover, as Figures 4.9-c and 4.9-f show, the improvement applied to HTSS.3 significantly reduces the latency from the point that the new tasks are introduced in the pipeline until they are ready. Processing a finished task in the three prototypes takes the same number of cycles. This is the reason that the difference between HTSS.3 and the two other prototypes for processing five dependent tasks (i.e., 253 vs. 298 and 305) is not as much as this difference for processing five non-dependent tasks (i.e., 98 vs. 242 and 230).

4.4 Summary

In this chapter the VHDL implementations of the three hardware prototypes of hardware Task Superscalar architecture are described. The packet processing latencies obtained for each hardware module are presented along with the preliminary performance results. Those results are used in the subsequent chapters to perform a design space exploration of the HTSS.3 prototype.

Chapter 5

Design Space Exploration of HTSS

The objective of this chapter is to perform a hardware design space exploration of HTSS. To this end, first, the implemented software cycle-accurate simulator, called SimTSS, is described. Then, the methodology, experimental framework and benchmark applications used for design space exploration are described. The results obtained for different HTSS configurations that provide maximum performance with minimum number of components and minimum memory capacities for different processing environments are presented. Finally, the HTSS architecture and its software equivalent, the Nanos++ runtime system, are compared.

5.1 Simulator for the Design Space Exploration

In order to perform a design space exploration of the HTSS architecture a cycle accurate software simulator, called SimTSS, has been designed. In this section, the design of the software simulator as well as its workflow are explained. After that, the methodology, experimental setup and benchmark applications which are employed for this thesis are introduced.

5.1.1 Simulator Description and Usage

Figure 5.2 illustrates the high level structure of SimTSS. SimTSS is a tiled pipelined architecture that consists of one iGW, n iTRSs, n eORTs, one TS and a network configuration including several arbiters and FIFOs that connect all the modules. The SimTSS workflow is designed based on HTSS.3 (see Sections 3.1.4 and 4.1.3 for a detailed description of HTSS.3). In addition, SimTSS is a configurable architecture that accepts several parameters to simulate a specific HTSS design. These parameters include the number of TRSs, the number of eORTs, the number of entries in every memory and some configuration parameters of dependence memory (DM), that has a set-associative structure.

Similar to HTSS prototypes, components of SimTSS communicate to each other using packets (message passing communication). Each component uses at least one finite state machine (FSM) for processing input packets and producing output packets, as well as accessing to the memories. The components of SimTSS are interconnected by several FIFOs and arbiters that are scaled in accordance with the rest of the modules. Although, the interconnection network of SimTSS is further configurable, we have adapted it to the network structure of the real HTSS prototypes. The network configuration (i.e., FIFOs and arbiters) decouples the work in the modules reducing stalls in the system.

The amount of parallelism that the Task Superscalar pipeline can uncover depends on the capacity of the memories which are used for storing task information, their dependences and the versions of these dependences. On the other hand, the performance of the Task Superscalar architecture depends on both the capacity of the memories and the number of components.

The Task Superscalar architecture has three types of memory:

- TM (task memory) for saving in-flight tasks and their dependences,

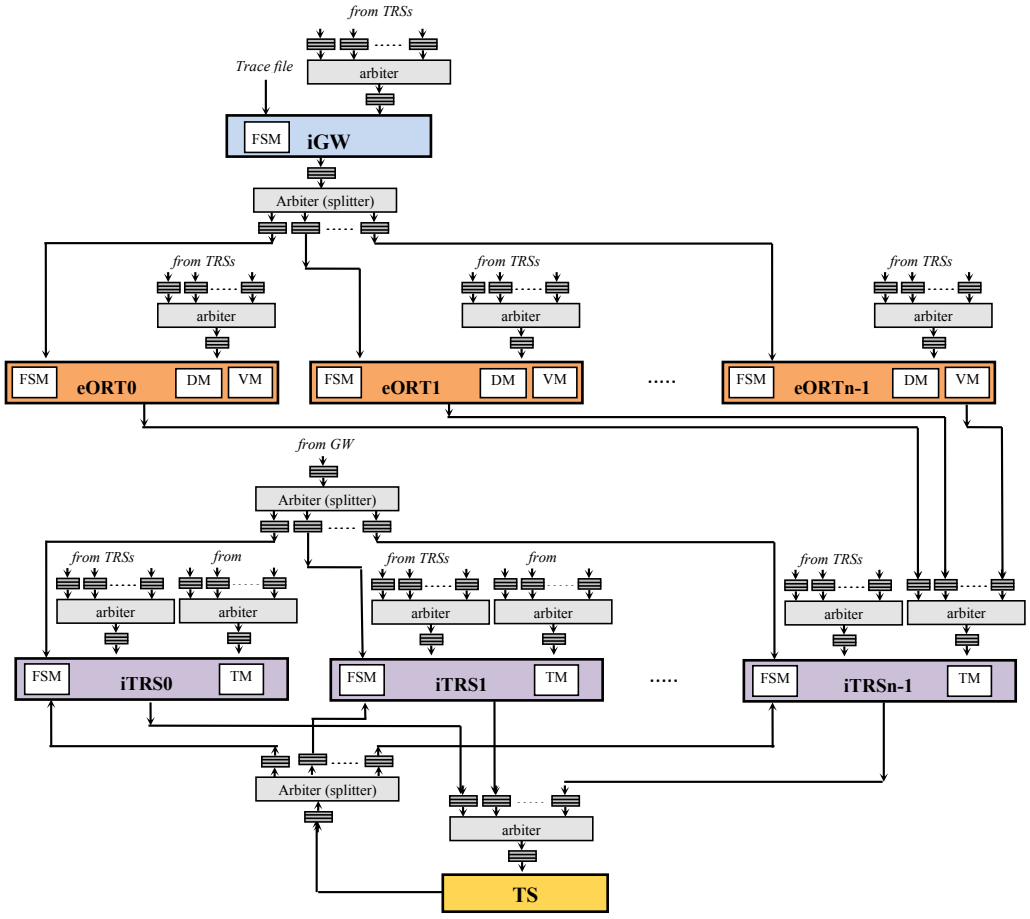


Figure 5.1: High level description of SimTSS

- DM (dependence memory) for keeping dependences and perform dependency analysis,

- VM (version memory) for saving different versions of the above dependences.

The TM is embedded into the iTRS module while both DM and VM are placed in the eORT module. The TRSs keep the information of all alive tasks in the TSS pipeline that are called in-flight tasks. In-flight tasks comprise tasks waiting for data dependency analysis, ready tasks waiting to be sent for execution, tasks being executed, and finished tasks that are still pendent to delete from the system. However, it should be emphasized that iTRSs storage size does not exclusively determine the effective number of in-flight

tasks, as this number might be limited by the number of entries of VM and DM.

The eORTs maintain an entry for each memory object used by in-flight tasks in the DM, and the corresponding version(s) of that objects in the VM. As such, the number of entries they can store affects the number of in-flight tasks.

Table 5.1 lists the input parameters of SimTSS. In SimTSS, we can select any number of TRSs and eORTs, as well as any number of entries for the different types of memories (i.e., TM entries, DM entries and VM entries). In addition, DM has two more parameters to be determined: the number of ways and the access mode. Access mode determines whether DM uses the bit based mapping (standard) hash function or an improved hash function for distributing dependences in DM. As it will be shown later on, the hash system is critical in the system as it significantly reduces system stalls caused by several dependences (more than the way size) with the same index trying to be placed in the same DM set (DM memory conflicts).

Table 5.1: Parameters of SimTSS

SimTSS Parameters	Description
# TRSs	Number of TRSs that can be any number between 1 to n
# eORTs	Number of eORTs that can be any number, power of two, between 1 to n
TM entries	Number of in-flight tasks per TRS
VM entries	Number of versions that the system can keep per eORT
DM entries	Number of dependencies that the system can keep per eORT
# ways	Number of ways in each set of the DM (Associativity of Dependence Memory)
DM access mode	Selecting standard hash or the improved SimTSS hash
# workers	Number of available workers

The configurable simulator allows to deeply evaluate the number of components and memory entries of HTSS suitable for different systems. Therefore, it can be found a HTSS configuration with the minimum number of components and memory entries that provides maximum performance for a given amount of processing resources.

As explained in previous chapters, each component of hardware Task Superscalar architecture has been implemented in VHDL. Using the HDL RTL simulator, the latency of each component for every state of their FSM has been obtained. Those numbers have been used to tune the simulator with the same behavior as the real machine. Then, the simulator has been used to do the design space exploration as it would have been very

complicated and time consuming to design a synthesizable hardware Task Superscalar with a configurable number of components and memory modules entries in VHDL, and then examine it with some HPC applications.

SimTSS gets an input configuration file which initializes its parameters (i.e., number of workers, number of TRSs, number of eORTs and number of memory entries and parameters), and selects an input trace. The input trace includes data and meta-data of tasks (TaskID, execution cycles, and number of dependences, direction of dependences and issue time of each task) that are obtained by instrumenting the source code of the applications. When the execution of a trace is completed, SimTSS reports the time at which each task finishes, total number of required cycles and some statistics that have been used in the design process.

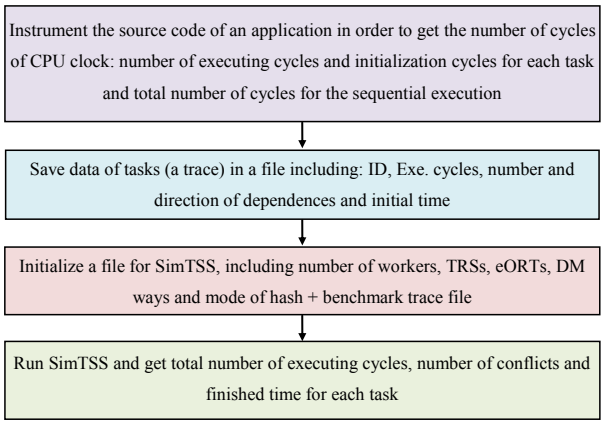


Figure 5.2: Workflow of SimTSS usage

For generating the traces, the cycle counters of the target processor have been used in order to find out the cycles required for executing each task, the cycle in which the task was issued and the total number of cycles for completing all tasks of an application. In fact, the total number of cycles of executing tasks is equal to the sequential execution of the application. Note that the number of cycles used for instrumentation instructions is not accounted in the total number of cycles.

5.1.2 Methodology

As it is mentioned in the previous section, SimTSS has several input parameters (e.g., number of modules and number of memory entries), that can be determined by the user. Modifying the parameters of SimTSS, different configurations have been examined to find the best configuration of HTSS that provides highest performance for a given number of processors (workers) while employing minimum hardware resources. Also the effect of pipeline parallelism (i.e., number of TRSs and eORTs) has been explored for each benchmark considering different number of workers. The distributed design of the pipeline facilitates speeding up the overall decode rate, by overlapping the different works of managing task dependencies and decoding tasks. Specifically, replicating the eORTs enables multiple dependencies to be recorded in parallel, whereas TRS replication reduces the per-TRS load and distributes the TRS loads, and thereby increases the overall processing rate of inter-TRS communication.

The study is divided into three analytical phases: In phase one, it is assumed that there are unlimited-resources available for prototyping HTSS. Based on this assumption, a design with more-than-enough number of components and more-than-enough number of memory entries is used. This configuration is called big configuration (BigConf). With this configuration, first, the minimum number of workers, *Wmin*, that provides maximum performance is determined for all the benchmarks. Then, using this number, the minimum number of entries for each memory and the minimum number of modules that allow to obtain this maximum performance is determined. Using Wmin workers, first the effect of different numbers of entries of the task memory (TM entries) is studied and fixed. Using Wmin workers and TM entries, good design points for VM entries and number of TRSs are found. Then, the effect of different DM parameters (i.e., hash, DM entries, and DM way) on performance is analyzed. At the end of this phase, a design configuration which provides high performance is obtained. This configuration of HTSS is called *high performance computing configuration* (HPCConf).

In phase two, the capabilities of the HTSS with minimum resources are explored. Here, a HTSS with only one TRS and one eORT is assumed. This configuration is explored with different number of workers and different number of entries (length) for memory modules in order to obtain a selected memory configuration suitable for small quantities of workers. This configuration is called *minimum configuration* (MinConf).

Using the results of phase one (BigConf and HPCConf) and phase two (MinConf), in phase three different aspects of HTSS are studied, compared to the *Parallelism* and *ZeroHTSS* systems. In the *Parallelism* system, an ideal design with any number of components and memory entries, that provides the highest possible speed up, is used to show the performance of the system compared to the potential one for a given application. The performance provided by the *Parallelism* system does not depend on the number of workers or the number of HTSS components and memory entries. It only depends on input trace and, in fact, the critical path of an application limits the *Parallelism* system to provide infinite performance. On the other hand, the *ZeroHTSS* system is a HTSS with unlimited number of resources, where packets are processed in each FSM of the component immediately (in zero cycles). Those two ideal approximations, the *Parallelism* and the *ZeroHTSS* systems, will be used to highlight the strength of the HTSS system and provide some insights for future improvements.

Finally, a comparison against Nanos++ runtime system is performed with realistic applications that will show the potential of the presented designs in real environments.

5.2 Frameworks and Benchmarks of the Design Space Exploration

5.2.1 Experimental Setup

In order to evaluate the capabilities of the HTSS, a group of real applications has been selected. All the applications can be obtained from the BSC Application Repository (BAR) [146]. The applications are annotated with OmpSs programming model pragmas to determine the tasks and their dependences. For every task, one pragma specifies the dependences and their directions (i.e., input, output and inout). With this information the source code of the selected applications has been instrumented to generate the traces used as input data to SimTSS.

On the other hand, the results of the real software runtime system of OmpSs applications have been obtained using the same task decomposition strategy. The OmpSs implementation used is based on Mercurium 1.99 source to source compiler [128], and Nanos 0.7a runtime system [124]. The applications have been executed (sequentially and in parallel) in a shared memory machine node with 2 NUMA nodes with 1 socket each. Each socket is a Xeon E5645 with 6 cores each at 2.4 GHz. The system has 24 GB of RAM Memory. The L1 memories (data and instructions) have 32 KB and the L2 has 256 KB per core. The system has also a shared L3 (for each processor) of 12 MB.

5.2.2 Benchmark Applications

The benchmark applications employed in the evaluations are high performance computing (HPC) applications. The applications are described in the following subsections.

Applications descriptions

Cholesky Factorization. The Cholesky factorization decomposes a symmetric, positive definite matrix $A = LL'$, with L lower triangular. This implementation divides A into b×b blocks or m×b panels. A block or a panel is divided in subpanels that contain t columns. This distribution includes three different variants of the Cholesky decomposition. `llchol` is the left-looking variant, as implemented by the routine DPOTRF() in LAPACK. `rlchol` is the right-looking version, as implemented by PDPOTRF() of

ScalaPACK. The aforementioned algorithms access A in blocks. The third variant, `prlchol` is a right-looking implementation, similar to `rlchol`, that uses panels instead of blocks. It resembles the naive right-looking implementation. The `llchol` variant was selected.

LU. The LU factorization decomposes an m×n matrix (m should be larger or equal n) $A = L^*U$, with L unit lower triangular (m×n) and U upper triangular (n×n). It is typically used in the solution of systems of linear equations. This implementation mimics the operations from LAPACK's DGETRF(). The matrix A is divided into column blocks or panels. This package includes three variations on this general theme. `lu` processes uniform-sized panels from left to right. After the factorization of a panel, the updates are propagated to the panels to the left. `lull` is the left-looking version of `lu`. `lurecurs` implements a recursive algorithm, which potentially uses panels of different dimensions. The `lu` variant was selected.

Sparse LU Decomposition. This application as the above one, performs an LU decomposition, but over a square sparse matrix. The matrix is allocated by blocks of contiguous memory.

Heat diffusion. This is an implementation of an iterative solver for heat distribution. There are three user-selectable algorithms: Jacobi, Gauss-Seidel and Red-Black. The Gauss-Seidel method has been selected.

Traces descriptions

Table 5.2 presents information about the characteristics of the benchmark applications. It includes the input configuration, number of tasks, and average of task sizes as well as maximum and minimum task size for each benchmark. In addition, it presents average number of dependences, average tasks distance and sequential execution cycles.

Input configuration of the applications in Table 5.2 states the parameters that have been chosen for running each application. It is important to mention that the configuration has been selected for stressing HTSS system. Those are not the best ones to solve the problem in a sequential system, but generate several small tasks issued as fast as possible. In other words, the configuration parameters of the benchmarks were chosen trying to obtain executions that generate several fine-grained tasks. The execution time obtained for the given problem size was not a concern in this point as the experiments try to measure the ability of the hardware to manage tasks. The size of a

Table 5.2: Information of benchmark traces

Application	Cholesky	SparseLU	Heat	LU
Input configuration	100 , 2	64 , 8	256 , 32	256 , 1
Number of tasks	22100	11472	1025	32896
Average task size (in cycle)	778	9835	1116	1970
Max task size (in cycle)	84704	147148	3124	229924
Min task size (in cycle)	416	3344	24	488
Average number of dependencies	2.88	2.90	4.996	2
Average task distance (in cycle)	31.06	139.06	39.63	24.78
Sequential execution cycles	19942850	114419887	1184928	65601061

task is measured as the number of cycles that are used for executing it. This number directly depends on the amount of computation that should be carried out for each task. Task distance is the number of cycles between the arrivals of two successive tasks. In the table, the average of tasks distance is presented. Sequential execution cycles states the number of cycles required in the sequential execution.

Here, the different input configurations are described. Those configurations have been selected in order to have large number of small tasks that are issued close to each other. For Cholesky, 100 is the size of the matrix, and 2 is the size of the block. Therefore the number of blocks is equal to 50. For SparseLU, the first number is the number of blocks in each matrix dimension. In Table 5.2, the matrix has 64×64 blocks. The second number of this application is the block size in each block dimension (8×8 blocks in the case of the table). For LU, the first number is the number of rows and columns of the matrix (dimensions of the matrix) and second one is the number of columns in a panel. In the case of Heat, the first number determines the matrix size and the second one is the number of blocks in each matrix dimension.

The number of tasks and average of task sizes as well as maximum and minimum task size for each benchmark are presented. As it can be seen in Table 5.2, Cholesky has a lot of small tasks. In fact, most of its tasks are computed in less than 1000 cycles and it has only two big tasks (more than 3000 cycles). Cholesky tasks have 2 or 3 dependencies each. Only two of them have one dependency. In total, 22100 tasks have 63750 dependencies.

In the case of LU, most of the tasks sizes are between 1000 and 3000 cycles. Few of

them last less than 1000 cycles, and only two of them last more than 3000 cycles. Of these, one is 111896 cycles long and the other lasts 229924 cycles. They are the first two tasks of the trace. The output dependence of the first one (Task-0) is consumed by the 255 sequent tasks (i.e., Task-1 to Task-255), and the output of the second big task (Task-1) is used by the 255 next tasks, from Task-256 to Task-510. This behavior is followed by subsequent tasks, but, they are not so big. Therefore, in LU application many tasks (every 255 tasks in our selected trace) are dependent to one task. Every task in LU has two dependences, one input and one inout. In total, the selected trace contains 65792 dependences. Compared to the other benchmark applications, SparseLU has larger tasks. Most of SparseLU tasks have a size between 8000 to 11000 cycles, and the rest except one are smaller than 8000 cycles. The SparseLU has only one big task with a size of 147148 cycles. SparseLU has tasks with one, two or three dependences, but most of them have three dependences. In total, the SparseLU trace has 33296 tasks.

For Heat trace, all the tasks are almost of the same size; between 1000 and 3000 cycles, except one which only lasts 24 cycles. In this trace all tasks have five dependences, one inout and four input dependences; hence it has 5121 dependences for 1025 tasks.

We compute average tasks distance as *((initial time of last task - initial time of first task)/number of tasks)*. As average task distance shows, compared to the average task sizes, the tasks are really close to each other, especially for Cholesky, LU and Heat applications. On the other hand, as SparseLU operates on a sparse matrix, it has a greater number of cycles between two successive tasks.

Finally, the sequential execution cycles number is a baseline number for comparing results of SimTSS with. For SimTSS analysis, speed-up results are presented. Those speed-ups are obtained by comparing the sequential baseline execution of an application to the simulation of the input trace of that application, with the same application parameters (i.e., speed-up=sequential execution cycles/SimTSS execution cycles). Note that each task execution in the simulator lasts the same number of cycles than the sequential task execution.

5.3 Design Space Exploration of HTSS

In this section, a deep design space exploration of the HTSS performed using the software simulator (SimTSS) is presented. The main goal is to explore which amount of resources (number of TRSs, number of eORTs, capacity of TM, VM and DM and also suitable structure of DM) is necessary in order to be able to fully exploit current and future many-core designs. For this, the results obtained from running the benchmarks are presented and evaluated in order to find a suitable configuration of the hardware Task Superscalar prototype for different performance configurations.

5.3.1 HTSS for High Performance Computing

To start the design space exploration, a very big configuration of the HTSS system with more-than-enough resources, presented in Table 5.3, has been selected. For each configuration parameter, the more-than-enough resources number is selected in such a way that even if the value is halved the time results of the system remain the same. The more-than-enough configuration is called as BigConf.

Table 5.3: Configuration of a HTSS with more-than-enough resources

SimTSS Parameters	Value
# TRSs	32
# eORTs	32
TM entries	16K
VM entries	16K
DM entries	16K
# ways	16

Using BigConf, the number of workers that provide maximum speed-up can be determined. To do this, first, the minimum number of workers that provides the highest speed-up for each benchmark is found, and then the maximum number across all the applications is selected.

Figure 5.3 presents the speed-up (Y-axis) obtained executing the traces of the four benchmark applications on SimTSS with maximum resources for different number of workers (X-axis). As Figure 5.3 shows, the speed-up increases by increasing the number of workers. Of course, the diagrams of speed-up become flatten when we reach a certain

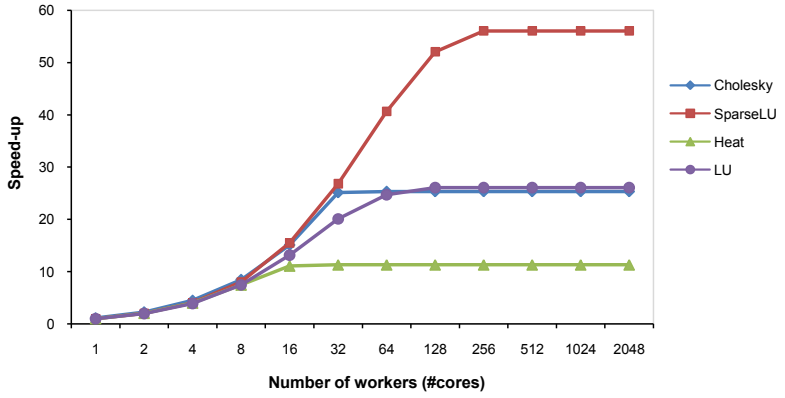


Figure 5.3: Summary of speed-up of a HTSS with more-than-enough resources

number of workers; this number is 64 for Cholesky, 256 for SparseLU, 32 for Heat and 128 for LU.

TRS parameter configuration

Once an upper limit to the design results has been established, it is a matter of reducing the number of resources in order to obtain a configuration with an affordable amount of resources suitable for HPC systems with 256 processors. First of all, the effect of changing the number of entries in the TRS memory (that is, limiting the maximum number of in-flight tasks that the system supports) is shown when HTSS has only one TRS. All the other parameters are maintained as in the more-than-enough configuration.

The summary of results is shown in Figure 5.4 where the Y-Axis shows the speed-up over the sequential execution as a function of the number of entries in the Task Memory (X-Axis). In this figure, two interesting effects can be observed: the first one is that for the selected traces 512 in-flight tasks seem enough when the system has only one TRS. The second observation is that the speed-ups decrease compared to those observed in Figure 5.3 due to the effect of having only one TRS module in the system. This happens, in particular, for the Cholesky benchmark. For other benchmarks, the speed-ups remain very similar to previous results. Regardless of the TM size, the time that the TRS uses to process the tasks may become the bottleneck of the system. That can be solved increasing the number of modules of HTSS, hiding the latency of the TRS processes.

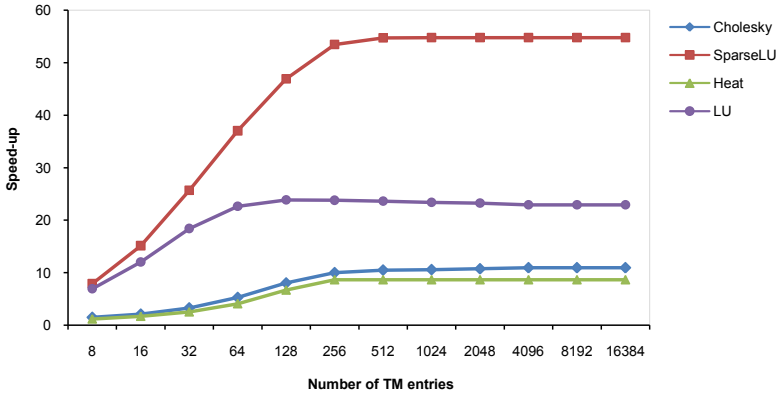


Figure 5.4: Speed-up obtained as a function of the number of Task Memory entries

Table 5.4 shows how changing the number of TRSs and its memory size influences the number of cycles for the Cholesky application. As it can be seen in the table, the optimum design point is to have eight TRS modules with the capacity to store 512 tasks each (for a total of 4K in-flight tasks), for the Cholesky benchmark. However, that configuration is only ideal for the specific case of Cholesky and presents serious drawbacks from the hardware resources point of view: eight TRS modules represent a large interconnection network and, furthermore, 4K in-flight tasks demand roughly 240KBytes of memory storage for the tasks and more space in the other memories that should be scaled accordingly. Considering both the results of Figure 5.4 and Table 5.4 for all the benchmarks and the hardware resources requirements of an eight-TRS configuration, the selected prototype has been limited to four TRSs with a 256-entry TM each, reducing the interconnection network and memory requirements, while guaranteeing high speed-up. Similar tables to Table 5.4 are presented for the other benchmarks in appendix D. As it can be seen from these tables, the other three applications are less demanding than Cholesky regarding the number of TRSs and TM entries. Therefore, the results obtained from Cholesky are good for all the applications.

Table 5.4: Speed-up of Cholesky application as a function of number of TRS modules and their memory size

TM entries	1 TRS	2 TRSs	4 TRSs	8 TRSs	16 TRSs
8	1.52	2.14	3.36	5.56	9.38
16	2.12	3.35	5.55	9.37	15.10
32	3.31	5.50	9.33	15.07	21.61
64	5.30	9.08	14.99	21.56	23.74
128	8.06	13.78	21.34	23.77	24.60
256	10.03	17.11	**23.45**	24.59	25.32
512	10.51	17.69	23.97	*25.31*	25.32
1024	10.60	17.91	24.92	25.31	25.32
2048	10.77	18.33	25.30	25.31	25.32
4096	10.97	18.61	25.30	25.31	25.32

eORT parameters configuration

After selecting the TRS modules configuration, the next step is to select a good eORT modules configuration. This work is more difficult as the eORT module is more complex with two different memories and its effect on the system performance is not so obvious. As explained before, the eORT modules keep track of the dependency chain and to do so, they have to store both all the dependencies of all the tasks in the DM and all the versions of those dependencies (the different values that the dependency can have due to the different in-flight tasks that produce this value) in the VM.

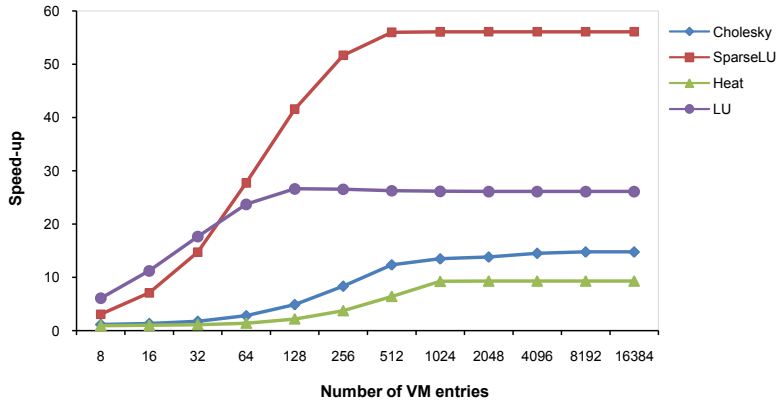


Figure 5.5: Speed-up obtained as a function of the number of Version Memory (VM) entries

The VM is the most simple. Like the TM, it is used in an indexed way (i.e., any value can be stored in any entry) with direct memory access. Indeed, for the VM, the only parameter that may influence the performance of the design is the capacity of the VM (#entries) since a new version can be assigned to any empty entry. Figure 5.5 shows the speed-up obtained for each benchmark when the number of entries is modified and we have only one eORT module and more-than-enough TRSs (32 TRSs). As it can be seen in the graph, the Cholesky application is, as in the case of the TM, the most demanding one, needing 4096 entries in the VM to achieve the peak performance.

Considering that number of entries, we show in Figure 5.6 how changing the number of eORT modules affects the speed-up when the total number of VM entries is maintained constant and the Dependency Memory (DM) is keep at its more-than-enough value. In particular, it can be observed that four eORTs with 1024 entries each (for a total of 4096 entries) achieves the upper limit of the performance.

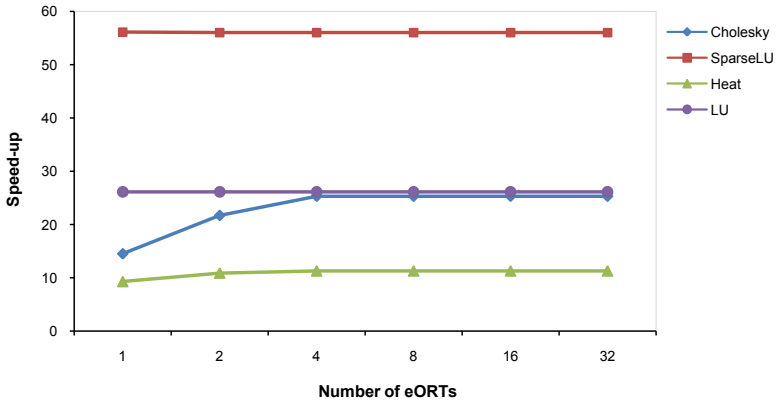


Figure 5.6: Speed-up obtained as a function of the number of eORT modules

The Dependence Memory (DM) is the key element in the eORT module. It keeps track of all the dependencies of all the in-flight tasks in the system. When a dependence enters the system, it should efficiently find if the dependence is new or not and update its meta-data correspondingly. As the latency of this search is critical, the ideal way to store the dependence information would be in a direct mapped memory. However, the DM is not a cache and when a block in the DM is full, the system can not replace and flush the existing entry. Instead, it should stall and wait (maybe for a while) until the dependence that uses the same entry is no longer alive. The reason is that the

dependency identifiers (addresses of dependences) are unrelated because of the data alignment in the application tasks. Therefore, when storing them in the corresponding DM, several consecutive dependences may be stored in the same entry of DM. As the DM can not discard entries, until the previous dependence is deleted from the pipeline when all the tasks that use it finish, the requester dependence(s) and its tasks have to wait. This may cause large stalls if the DM is implemented as a direct mapped memory or if the hash function doesn't appropriately randomizes the addresses of the dependences. For this reason, an associative memory, and a more complex hash function (Pearson-like hash [147]) than the usual (address less significant bits - LSB in Figure 5.7) is used to select the set.

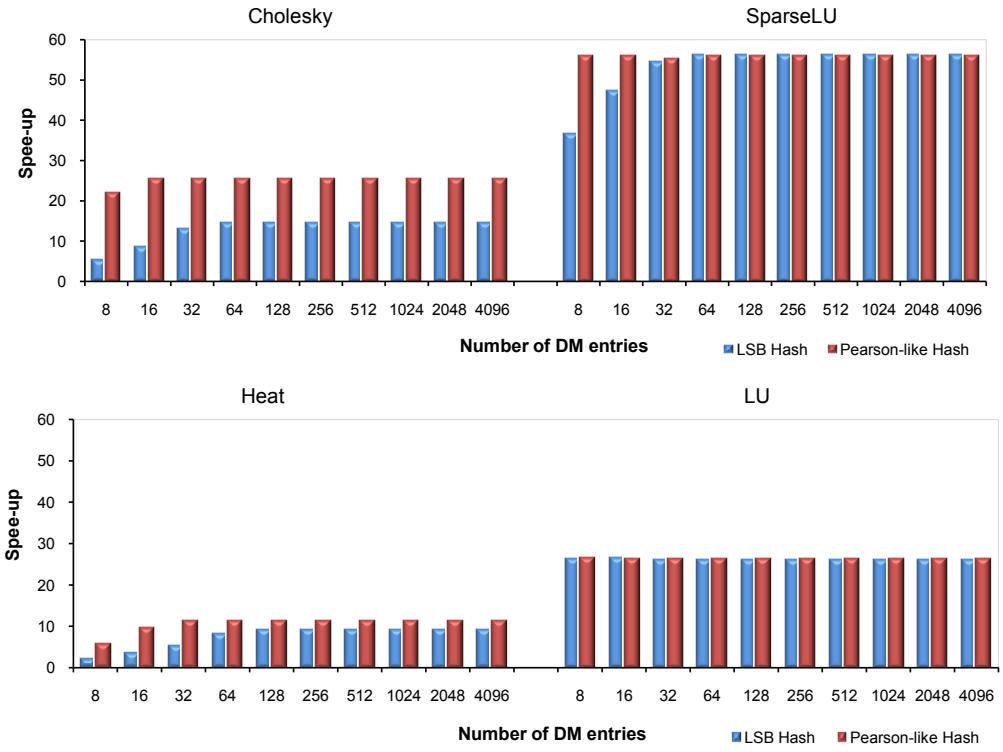


Figure 5.7: Speed-up obtained as a function of the DM entries with LSB and Pearson-like hash

Figure 5.7 shows the effect of the Pearson-like hash function in the speed-up obtained as a function of the number of entries in the DM. As it can be seen, the Pearson-like hash

function has better speed-up for all the cases or, from another point of view, allows the system to obtain the same results with a smaller number of DM entries. While the LSB-hash only uses the less significant bits of the dependence initial address to distribute them between eORTs and in the DM memory, the improved one uses Pearson-like hash to improve the distributions. The rest of the results presented in this section use the improved hash. Figure 5.7 also shows that 32 entries (with 16-way set associative each) for DM is enough to obtain the upper limit performance for the studied benchmarks.

The selected memory associativity is also key for the performance. The ideal would be having a full associative DM, but this is not possible in a real environment. The simulated DM has been designed so that the number of ways in a set can be configured. The effect of having different associativities with the Pearson-like hash has been studied and it is shown in Figure 5.8. This figure shows that using a larger number of ways results in higher speed-up maintaining the total amount of memory. However, the higher speed-up does not come for free. For larger number of ways, more hardware resources and a more complex DM structure are required, so, eight-way has been selected as enough to provide good performance results while keeping the used resources affordable.

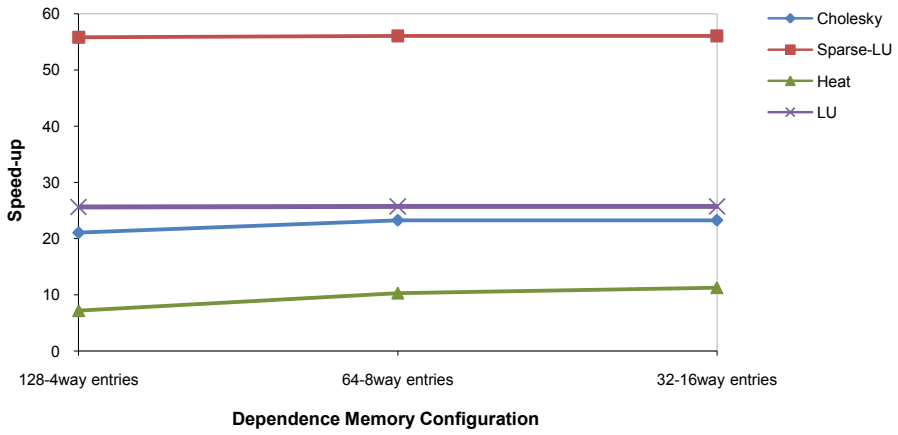


Figure 5.8: Speed-up obtained as a function of the associativity of the DM

Finally, with the selected hash, memory associativity and number of DM entries, we have performed a combined space exploration of the sizes of both eORT memories (DM and VM) as in a real execution their influence interacts.

In Tables 5.5 and 5.6, the speed-ups obtained by executing the benchmarks with SimTSS versus sequential execution are presented for different DM and VM sizes for each benchmark using a HTSS with four eORTs. In the tables, three values are pointed out: in blue the maximum speed-up of each application, in red the speed-up of the DM and the VM configurations that resulted from Figures 5.5 to 5.8, and finally, in highlighted yellow the speed-up of the definitive selected configuration obtained from these tables. As it can be seen, the previously selected number of entries for the VM (i.e., 1024) is more than enough and the more appropiate design point is 512 entries in the VM and 64 eight-way (512) entries for the DM of each eORT. Although this value is not the maximum for each benchmark, it is really close to the upper limit for all of them and represents an affordable amount of memory (3KB the VM and 5.25KB the DM) in each eORT module.

Our last experiment in the design space exploration is a crosscheck of the obtained values. The system has been evaluated changing the number of modules but maintaining the total amount of memory. The results of this experiment can be seen in Figure 5.9. As the figure shows it is necessary to have at least four TRS and four eORT modules to achieve the upper limit of the performance of the explored system. It also can be seen that more modules do not help to significantly improve that performance. This is due to the effect that having four modules is enough for exploiting the parallelism found in those benchmarks and increasing this number only results in a more complex network. Note however, that doubling the number of modules halves the memory in each one of them. This doesn't influence the capacity of the system in terms of tasks as the TM is always fully occupied if there are enough tasks, but as every dependence can only be stored in the assigned eORT, halving their memories can sometimes result in some stalls if their occupancy is not perfectly balanced.

In conclusion, the proposed configuration for a HTSS machine is composed by ten modules: one Gateway, four TRSs, four eORTs and one TS. Each TRS has a 256-entry TM. Each eORT module has 2 memories: the VM is an indexed array of 512 entries while the DM is an eight-way set associative memory with 64 entries (to also amount a total of 512 entries). We call this configuration HPC configuration of HTSS (HPCConf) that provides maximum speed-up while utilizing a minimum amount of resources for up to 256 workers.

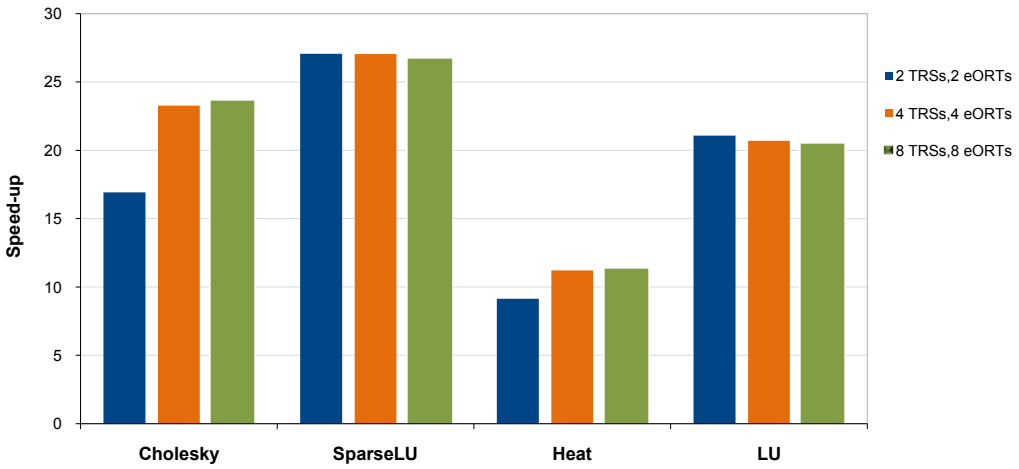


Figure 5.9: Effect of changing the number of modules maintaining the sizes of the memories

Table 5.5: Speed-ups of simulating the benchmarks with SimTSS vs sequential execution for a range of DM entries and VM entries with three pointed out speed-ups: in blue the maximum speed-up of each application, in red speed-up of the DM and the VM resulted from Figures 5.5 to 5.8, and in highlighted yellow the speed-up of the selected DM and VM configuration. a) Cholesky, b) SparseLU. (SimTSS configuration: four eORTs, 32 TRSs, 16K TM entries, eight-way DM with Pearson-like hash)

a) Cholesky		DM entries										
		8	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
VM entries	**8**	1,426	1,426	1,426	1,426	1,426	1,426	1,426	1,426	1,426	1,426	1,426
	16	1,709	1,907	1,926	1,925	1,925	1,925	1,925	1,925	1,925	1,925	1,925
	32	1,758	2,579	3,227	3,285	3,285	3,285	3,285	3,285	3,285	3,285	3,285
	64	1,758	2,657	4,466	6,402	6,759	6,762	6,762	6,762	6,762	6,762	6,762
	128	1,758	2,658	4,496	7,62	11,743	12,797	12,808	12,808	12,808	12,808	12,808
	256	1,758	2,658	4,496	7,624	12,573	19,189	20,858	20,858	20,858	20,858	20,858
	512	1,758	2,658	4,496	7,624	12,573	20,385	23,987	23,987	23,987	23,987	23,987
	1024	1,758	2,658	4,496	7,624	12,573	20,423	25,012	25,012	25,012	25,012	25,012
	2048	1,758	2,658	4,496	7,624	12,573	20,423	25,322	25,322	25,322	25,322	25,322
	4096	1,758	2,658	4,496	7,624	12,573	20,423	25,322	25,322	25,322	25,322	25,322
	8192	1,758	2,658	4,496	7,624	12,573	20,423	25,322	25,322	25,322	25,322	25,322

b) SparseLU		DM entries										
		8	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
VM entries	**8**	11,625	11,625	11,625	11,625	11,625	11,625	11,625	11,625	11,625	11,625	11,625
	16	11,697	19,557	23,636	23,681	23,681	23,681	23,681	23,681	23,681	23,681	23,681
	32	11,697	19,685	31,797	37,819	38,232	38,232	38,232	38,232	38,232	38,232	38,232
	64	11,697	19,685	32,128	43,935	49,635	50,140	50,140	50,140	50,140	50,140	50,140
	128	11,697	19,685	32,128	43,936	52,878	55,769	55,782	55,782	55,782	55,782	55,782
	256	11,697	19,685	32,128	43,936	52,889	56,055	56,051	56,051	56,051	56,051	56,051
	512	11,697	19,685	32,128	43,936	52,889	56,052	56,035	56,035	56,035	56,035	56,035
	1024	11,697	19,685	32,128	43,936	52,889	56,052	56,035	56,035	56,035	56,035	56,035
	2048	11,697	19,685	32,128	43,936	52,889	56,052	56,035	56,035	56,035	56,035	56,035
	4096	11,697	19,685	32,128	43,936	52,889	56,052	56,035	56,035	56,035	56,035	56,035
	8192	11,697	19,685	32,128	43,936	52,889	56,052	56,035	56,035	56,035	56,035	56,035

Table 5.6: Speed-ups of simulating the benchmarks with SimTSS vs sequential execution for a range of DM entries and VM entries with three pointed out speed-ups: in blue the maximum speed-up of each application, in red speed-up of the DM and the VM resulted from Figures 5.5 to 5.8, and in highlighted yellow the speed-up of the selected DM and VM configuration. c) Heat, d) LU. (SimTSS configuration: four eORTs, 32 TRSs, 16K TM entries, eight-way DM with Pearson-like hash)

c) Heat		DM entries										
		8	16	32	64	128	256	512	1024	2048	4096	8192
VM entries	8	1,034	1,034	1,034	1,034	1,034	1,034	1,034	1,034	1,034	1,034	1,034
	16	1,050	1,224	1,255	1,255	1,255	1,255	1,255	1,255	1,255	1,255	1,255
	32	1,050	1,242	1,800	2,014	2,019	2,019	2,019	2,019	2,019	2,019	2,019
	64	1,050	1,242	1,808	2,784	3,526	3,520	3,520	3,520	3,520	3,520	3,520
	128	1,050	1,242	1,808	2,787	4,231	6,161	6,211	6,211	6,211	6,211	6,211
	256	1,050	1,242	1,808	2,787	4,231	6,920	10,011	10,116	10,116	10,116	10,116
	512	1,050	1,242	1,808	2,787	4,231	6,920	10,353	11,314	11,310	11,310	11,310
	1024	1,050	1,242	1,808	2,787	4,231	6,920	10,353	11,306	11,313	11,313	11,313
	2048	1,050	1,242	1,808	2,787	4,231	6,920	10,353	11,306	11,313	11,313	11,313
	4096	1,050	1,242	1,808	2,787	4,231	6,920	10,353	11,306	11,313	11,313	11,313
	8192	1,050	1,242	1,808	2,787	4,231	6,920	10,353	11,306	11,313	11,313	11,313

d) LU		DM entries										
		8	16	32	64	128	256	512	1024	2048	4096	8192
VM entries	8	16,016	16,016	16,016	16,016	16,016	16,016	16,016	16,016	16,016	16,016	16,016
	16	16,975	21,484	22,181	22,194	22,194	22,194	22,194	22,194	22,194	22,194	22,194
	32	17,096	23,174	27,061	27,407	27,442	27,442	27,442	27,442	27,442	27,442	27,442
	64	17,096	23,175	27,08	27,033	26,356	26,329	26,329	26,329	26,329	26,329	26,329
	128	17,096	23,174	27,078	27,031	26,302	26,245	26,245	26,245	26,245	26,245	26,245
	256	17,096	23,174	27,076	27,024	26,295	26,185	26,185	26,185	26,185	26,185	26,185
	512	17,096	23,174	27,075	27,023	26,301	26,079	26,081	26,083	26,083	26,083	26,083
	1024	17,096	23,174	27,075	27,023	26,297	26,082	26,083	26,078	26,081	26,087	26,087
	2048	17,096	23,174	27,075	27,023	26,293	26,081	26,076	26,067	26,094	26,116	26,116
	4096	17,096	23,174	27,075	27,023	26,293	26,086	26,068	26,072	26,105	26,126	26,126
	8192	17,096	23,174	27,075	27,023	26,293	26,086	26,068	26,088	26,089	26,127	26,127

5.3.2 HTSS design with limited workers

In this section, the HTSS is analyzed with a more limited number of workers to evaluate the results obtained from the previous section, and find a suitable configuration for smaller systems like the ones that can be found nowadays. For this, it is assumed that the selected design has only 32 available workers and a HTSS with four TRSs and four eORTs. The goal is to repeat the study of the effect of the memory sizes on performance but with limited resources in order to find out the minimum number of memory entries (i.e., TM entries, VM entries and DM entries) for a HTSS configuration for current systems (*CurConf*).

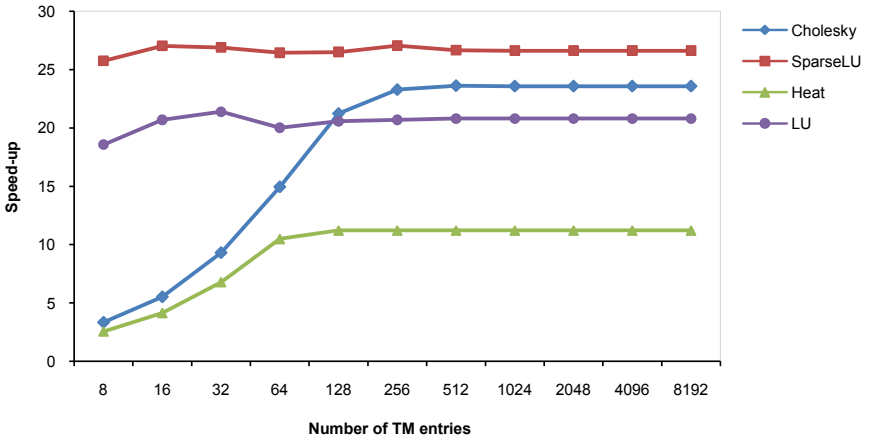


Figure 5.10: Effect of different number of TM entries on the performance of a system with 32 workers

Figure 5.10 shows the effect of different number of TM entries on performance. In particular, it can be seen that 256 entries are enough for the TM of each TRS when it is used in conjunction with a 64-element eight-way DMs with Pearson-like hash in each eORT.

Figures 5.11 and 5.12 show the same study varying the number of VM and DM entries, respectively. Both figures show the eORT memories effect on the performance of HTSS with only 32 processors. Results in Figure 5.11 indicate that 512 entries should be selected, as a good size vs. performance tradeoff for VM entries. This is also the same number of VM entries that was found in the previous study in Section 5.3.1. In

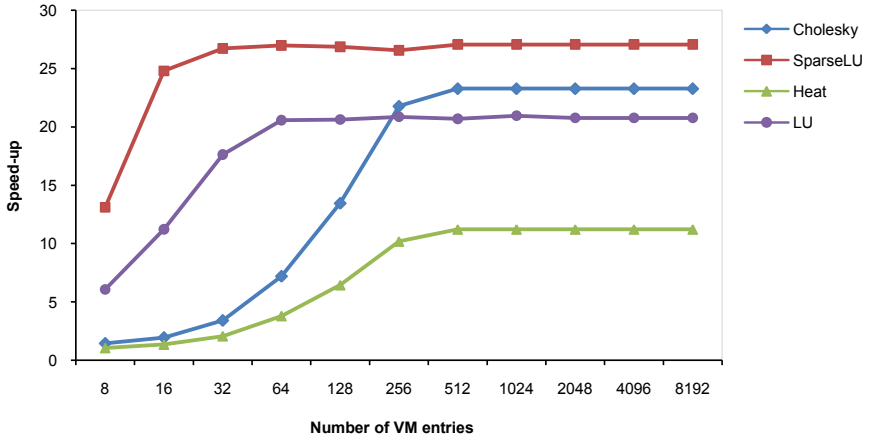


Figure 5.11: Effect of different number of VM entries on the performance of a system with 32 workers

the case of Figure 5.12, it can be seen that a 64-element eight-way DM with Pearson-like hash is enough for each eORT, also as found in previous analysis in Section 5.3.1.

As a result, the best configuration for using HTSS with a current system is four TRSs with 256 entries for TM, and four eORTs with 512 VM entries and 64 DM entries for eight-way DMs with improved hash. This architecture is able to store up to 1024 in-flight tasks in TRSs, 2048 versions and 2048 dependences in the eORTs. Note that the total sizes of the memories depend on the word-size of each memory.

The results show that the same hardware is necessary to manage 256 or 32 workers. Although this result seems counter intuitive, it is due to the fact that memory sizes are necessary to discover enough parallelism in the applications while the number of modules is necessary to keep pace with the task issue rate. Both factors depend mainly on the applications and not on the available number of workers. However, it can be seen in Figures 5.10 to 5.12 that with 32 workers the system gets a maximum speed-up of about 27x, while for 256 workers, a speed-up up to 56x for SparseLU can be obtained.

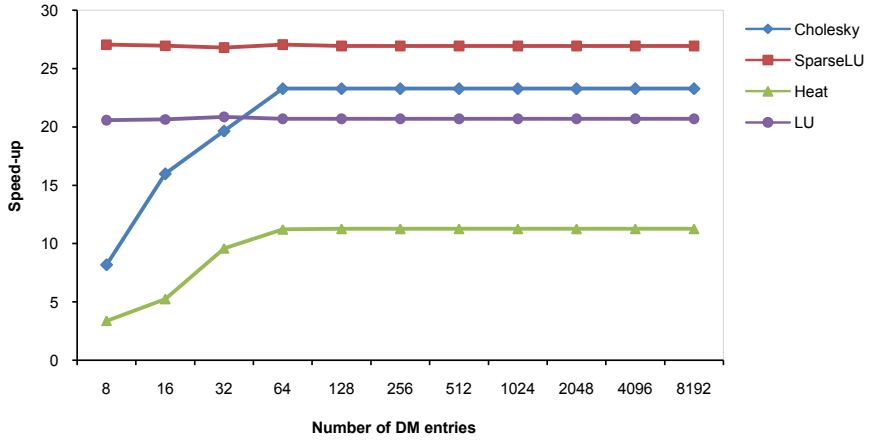


Figure 5.12: Effect of different number of DM entries on the performance of a system with 32 workers

5.3.3 Simple HTSS for Small Multicores

In this Section, a HTSS design with minimum resources is analyzed in order to find a configuration with minimum memory capacity that can provide acceptable speed-up for small or even embedded systems.

To do this exploration, six configurations of HTSS with only one TRS and one eORT have been selected. The first one, called Conf-FM (configuration of full memory), has an amount of memory with the capacity equal to the total capacity of the memories of HPCConf (i.e., 91.25 Kbyte). The second configuration, called Conf-FM/2, has half the memory amount than Conf-FM, and the third one has 1/4 the memory of the Conf-FM. This scale progresses until the sixth one that has 1/32 the memory of Conf-FM. The speed-ups that that design, with the different configurations, can achieve when a variable number of processors is used, have been obtained and are presented in Figure 5.13 for each of the applications. As the figures show, for up to eight workers, configuration Conf-FM/4 is the one with less resources that operates as well as the others in providing speed-up. When there are only four available workers, Conf-FM/16 configuration may be better than the rest as it uses even less resources to provide also near the same performance for this number of workers. As a result, Conf-FM/4 is selected as the minimum configuration (MinConf) for comparing it with the other selected HTSS configurations.

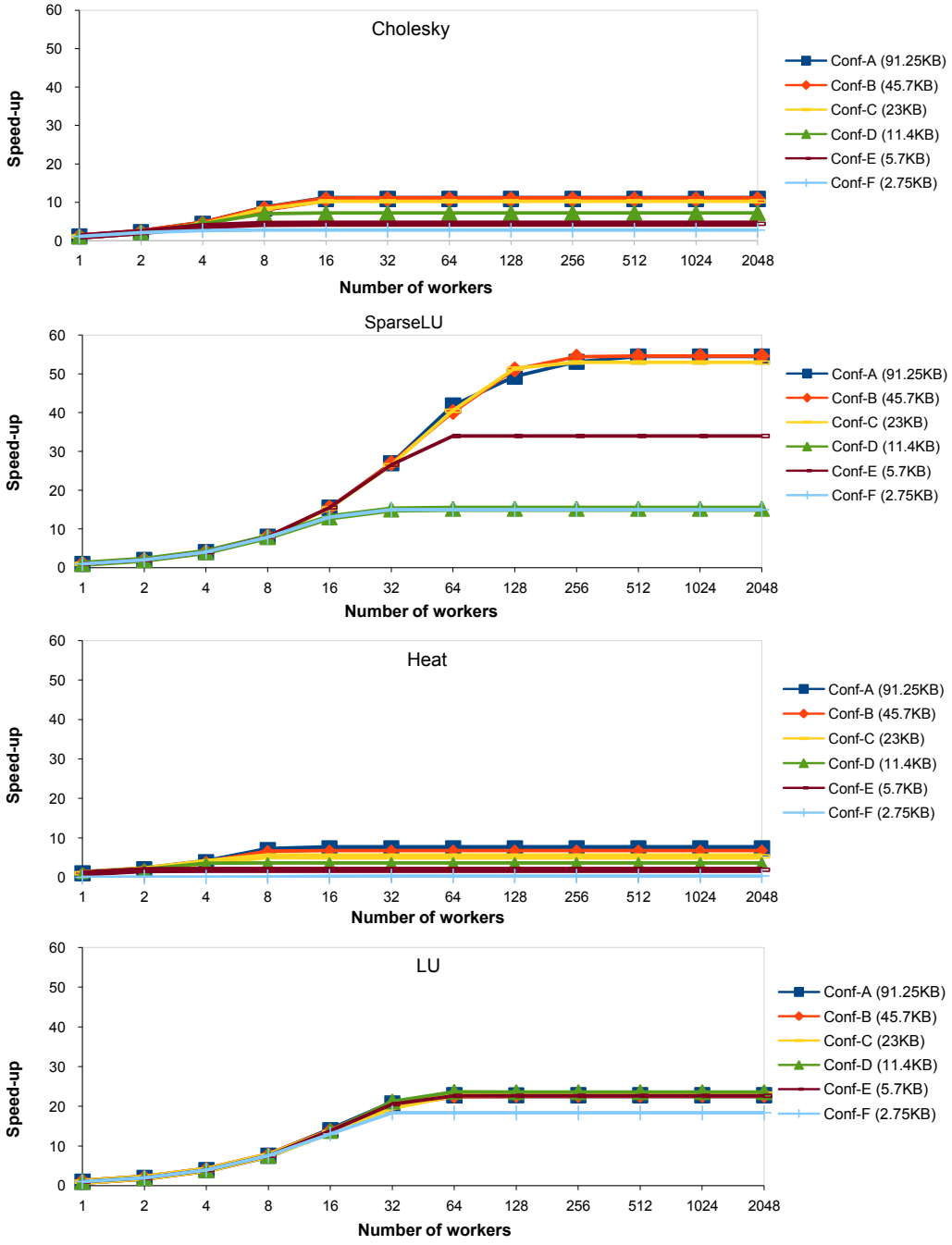


Figure 5.13: Speed-ups obtained with only one TRS and one eORT modules when changing the total memory sizes and the number of workers

5.4 Results of the Design Space Exploration

In this section, the selected different configurations of HTSS: HPCConf, MinConf and BigConf are evaluated and compared to *Parallelism* and *ZeroHTSS* systems in Figure 5.14 for analyzed applications. After that, HTSS and Nanos++ runtime system are also compared to each other.

5.4.1 Comparison of the Selected HTSS Configurations

To obtain a good idea of how well the proposed final systems behave, they have been evaluated with a different number of processors comparing the obtained speed-ups to two control configurations: *Parallelism* and *ZeroHTSS* systems. For each of the benchmarks, Figure 5.14 shows the maximum speed-up that can be obtained with the chosen parallelization strategy (*Parallelism* configuration) (*Parallelism*=$T_1/T_\infty$where T_1 is the sequential time and T_∞ is the time of the critical path in the parallel strategy supposing infinite resources) of a given benchmark. Figure 5.14 also shows the results that would be obtained with an HTSS that uses zero cycles to process any packet (*ZeroHTSS*), the results obtained with our previously commented HTSS more-than-enough configuration (BigConf), the selected configuration for big systems (HPCConf) and the selected minimum configuration (MinConf).

As it can be observed in Figure 5.14, the selected HPCConf performance is almost the same as *ZeroHTSS* system for all the benchmarks. Only for Cholesky a small slowdown can be appreciated as a trade-off of downsizing the resources. Also it can be seen that, as stated in Section 5.3.3, for systems with a small number of processors (up to eight) a minimum configuration (MinConf) is able to keep pace and so, it would be enough and affordable to be implemented in embedded systems.

Comparing the results shown in figure 5.14 to the *Parallelism* configuration, it can be seen that the implementable HTSS can extract all the possible parallelism for three of the four benchmarks once a certain number of workers is reached. The only exception is for the LU application which can obtain the maximum speed-up with the ideal (*ZeroHTSS*) implementation of HTSS but not with HPCConf or even with the more-than-enough BigConf. The difference in performance here is due to the large dependency chains of consumers (255 for each producer) that the LU application creates. Awakening 255 consumers means creating a sequential chain of 255 packets between the

TRSs and, consequently, when the last consumer is awakened several cycles have been wasted. To improve this, it can be proposed a system that simply creates a new version of a dependence when several consumers are detected. This new version awakens at the same time as the original one and splits the chain of packets into two different and parallel chains. However, this improvement has not been implemented as with more realistic task sizes this behavior will disappear hidden by the longer task execution times, as will be seen in Section 5.4.2.

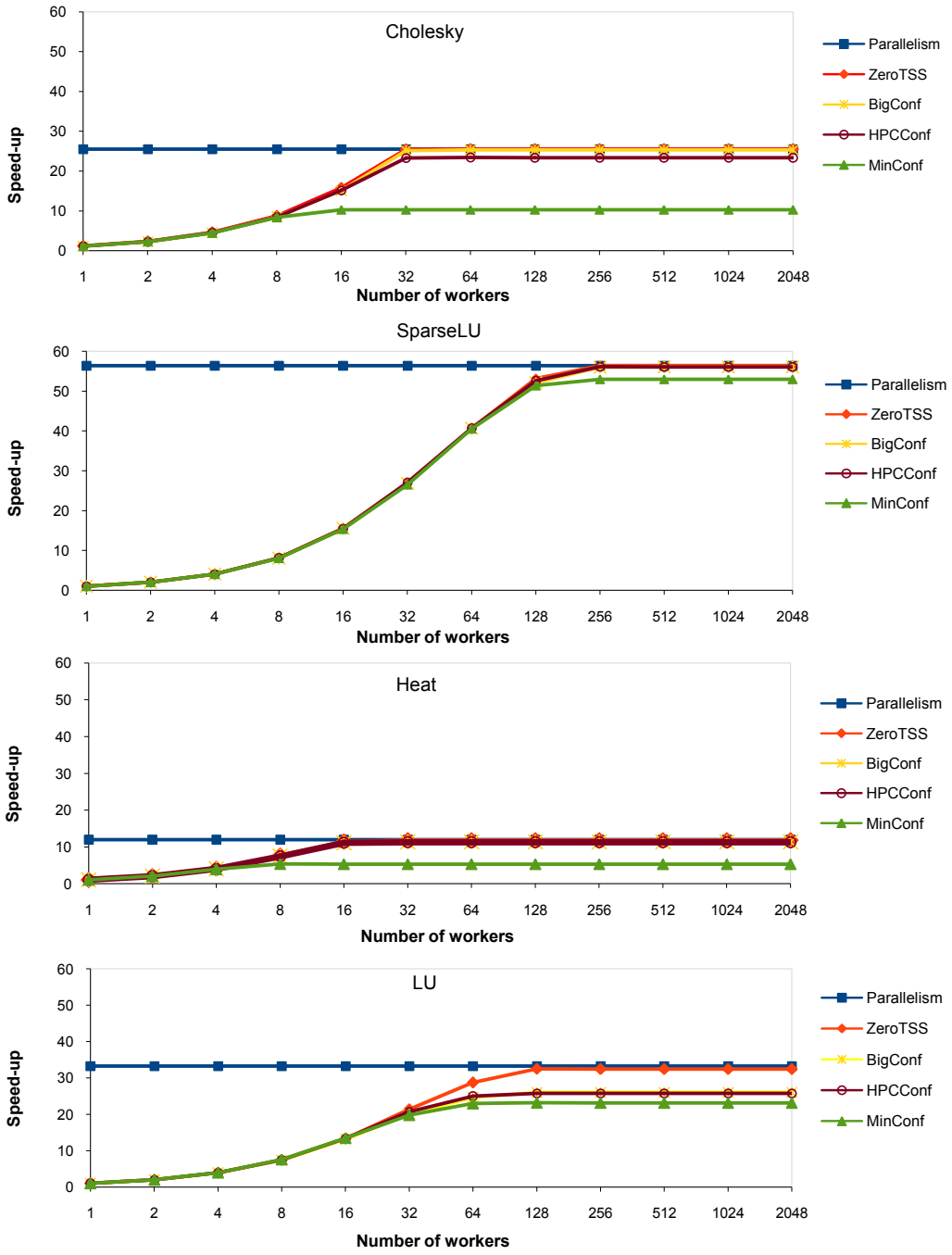


Figure 5.14: Speed-ups obtained for different number of workers with the *Parallelism*, *ZeroTSS*, BigConf, HPCConf and MinConf configurations

5.4.2 Comparison of HTSS to the Software Runtime Alternative (Nanos++)

In this section, the HPCConf configuration of HTSS is compared to Nanos++ runtime system for the benchmark applications. Figure 5.15 shows speed-up results of the benchmarks. Both HTSS and Nanos speed-up results are compared to the sequential execution of the benchmarks. HTSS results were obtained by simulation the benchmarks using HPCConf configuration, and the Nanos++ results were obtained running the parallel OmpSs version of the benchmarks. OmpSs runtime (Nanos++) is the software runtime alternative of HTSS. OmpSs (i.e., Nanos++) execution results are for a machine with 12 cores at 2.4 GHz (see Section 5.2.1 for more details). The Figure shows in the Y-axis the speed-ups obtained against the sequential execution when we change the number of threads (X-axis) and the parallel approach (the block size). Block size appears as a label beside the Nanos and HTSS labels in the legend of the figure. The executions shown in each graph solve the same problem for all applications: a 2048 problem size (matrix dimensions). To avoid the variability of comparing different executions, all the tests have been executed three times and the best results have been chosen. Also it is important to note that while Nanos++ real executions are influenced by the parallel memory behavior of the application, HTSS results are based in a sequential execution trace that can exhibit a different memory behavior.

In Figure 5.15, it can be seen that when the parallelism is increased (bars with diminishing block sizes) Nanos++ and HTSS take advantage of the increasing number of tasks (Cholesky 2048-1024 has only 4 tasks while Cholesky 2048-16 has 357760 tasks). However, as the task granularity diminishes (the problem size is the same in all the executions) the overhead introduced by the software runtime scheduler starts to introduce diminishing returns in the obtained speed-up. This effect can be observed in the last execution configurations in Figure 5.15: for Cholesky, bars 64-Nanos, 32-Nanos and 16-Nanos. The HTSS, on the other hand, can take profit of the parallelism of the application regardless of the parallelism granularity and, in fact, the more aggressive the parallelism, the better HTSS exploits it. This behavior is really desirable as it decouples the application parallelization approach from the hardware in which it is going to be executed, making parallel programmers' life easy.

In the case of LU in Figure 5.15, two interesting effects can be observed: First of all it shows super linear speed-up for the LU 2048-16 parallelization making Nanos++

perform better than the hardware. This effect can not be observed in HTSS as its results are extrapolated from the sequential execution but it will also occur in a real machine allowing the hardware to be at least as good as the software. The second effect that can be observed is that the delay introduced by the hardware when following large chain dependencies observed in Figure 5.14 has vanished due to the more reasonable size of the problem and the tasks. As it has been commented, this effect is easily solvable but the effort would probably not be worth in real implementations.

The Heat graph in Figure 5.15 shows the main limitation of HTSS. In this graph the speed-up decreases when the number of tasks increases (block sizes 16 and 8) for eight workers or more. The reason for this behavior is the limited amount of tasks that the HTSS can store. The Heat benchmark has a wavefront-like dependency pattern that only finds parallelism several tasks ahead of the current one. This number of tasks, that should be processed (not executed) before the parallelism is found, increases as the block size decreases and these problems exceed the task memory capacity of the prototype (1024 tasks). Of course this problem can be solved by simply increasing the total memory of the HTSS, but as the figure shows, even as it is, the hardware performs several orders of magnitude better than the software alternative. Indeed, a modification on the parallel implementation strategy of the wave-front pattern of Heat will overcome that problem without increasing the capacity of the memories of the hardware. The idea is to express the wave-front pattern with two nested loops: the outer loop will iterate among the diagonals (waves) of the wave-front pattern, and the inner loop will iterate on the elements of the diagonals. In that sense, all the elements of a wave will be near in time, overcoming the HTSS memory issue, and will be able to be run in parallel. Therefore, any wave-front problem that can be solved using that strategy will be highly efficient executed using HTSS.

The SparseLU graph in Figure 5.15 shows a behavior that resembles the Heat graph one. However, it is in part due to a completely different reason. In this application the HTSS with smaller tasks (8 HTSS bars) shows less speed-up than with bigger ones because the application does not scale properly for these parallelization approaches as the time required to divide the work exceeds the time to do the work itself. Consequently, there is nothing that can be done except changing the application code which is out of the scope of this work.

Figure 5.16 shows the number of task instance executions (right y-axis) and the average task size in cycles (left y-axis) of the executions in Figure 5.15 as a function of the block sizes (x-axis). As it can be seen in Figures 5.15 and 5.16, the software approach suffers not only when the tasks are small but also when the number of tasks grows exponentially. The hardware, on the other side, transforms its limited memory storage drawback in an advantage. HTSS keeps obtaining good results as it only maintains a limited number of in-flight tasks at the same time, but processing them very fast.

Another interesting side effect of using the HTSS instead of the software approach is that the hardware does not suffer from contention when the number of threads increases. This effect can be seen even for 12 threads compared to eight threads in the 256 bars of Cholesky in Figure 5.15. In those two bars, the number of threads augment and the HTSS can take profit of the increasing in available resources. Note that Cholesky 2048 with 256 block size has only 120 tasks and a maximum speed-up of 7.6× for this *Parallelism* strategy ($Parallelism=T_1/T_\infty$). However, the runtime is not able to do so, even obtaining fewer speed-ups with more resources. The reason for that different behavior is the decoupled design of the hardware that allows working in parallel in the different dependence chains that the application generates, avoiding contention caused by shared data structures.

In fact, taking this example of contention to the limit to better illustrate it, Cholesky 2048 with 64 block size has a maximum speed-up of 86× and the selected configuration can extract a speed-up of up to 72× with 256 workers. An even more parallel configuration (with eight TRS and eight eORT modules) with the same number of workers can scale up to a 83×.

This effect is fully shown in Figure 5.17 where it can be seen, for the chosen applications and the same problem size of 2048, the speed-up obtained when they are simulated in a system environment with 256 workers using the selected HPCConf. The speed-up is obtained comparing the simulation execution to the real sequential execution. For the sake of comparison, Figure 5.17 also shows the best speed-up that can be obtained for those traces with 256 workers ("Ideal 256" bars) and the improvement that will result when using 512 workers, both with a real doubled configuration (that is the same as the HPCConf but doubling the number of TRS and eORT modules, labeled "Double 512") and the ideal case (Ideal 512). Figure 5.17 shows that for all the benchmarks the selected configuration reaches speed-ups close to the ideal. For the most demanding

applications, those speed-ups can still be improved by simply increasing the number of modules in the system showing that even for very aggressive machines (larger than the one studied in this thesis) and demanding applications the decoupled HTSS system would be able to deal with the challenge.

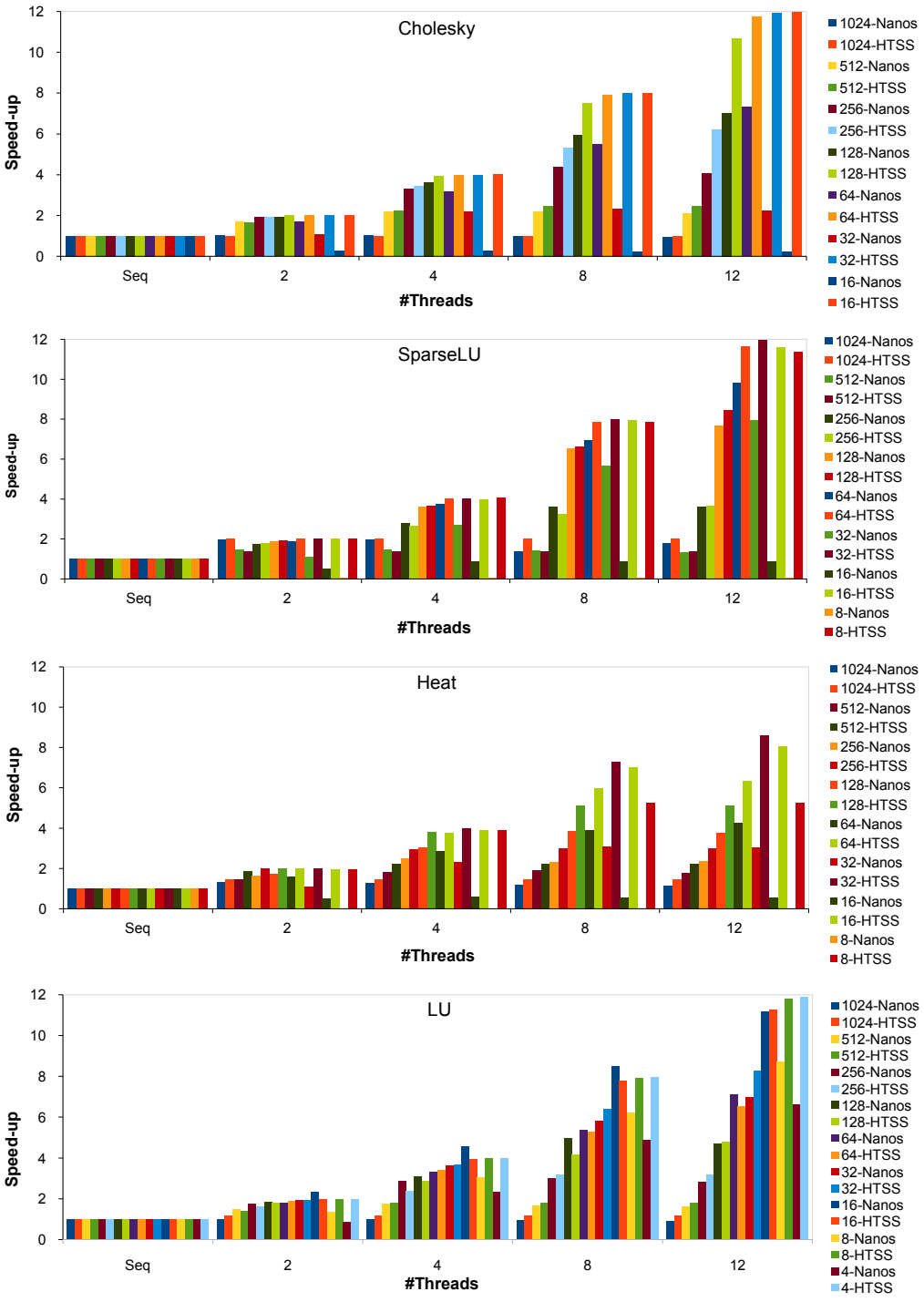


Figure 5.15: Comparison of Nanos++ and HTSS with different number of threads and block size for the same problem size (2048)

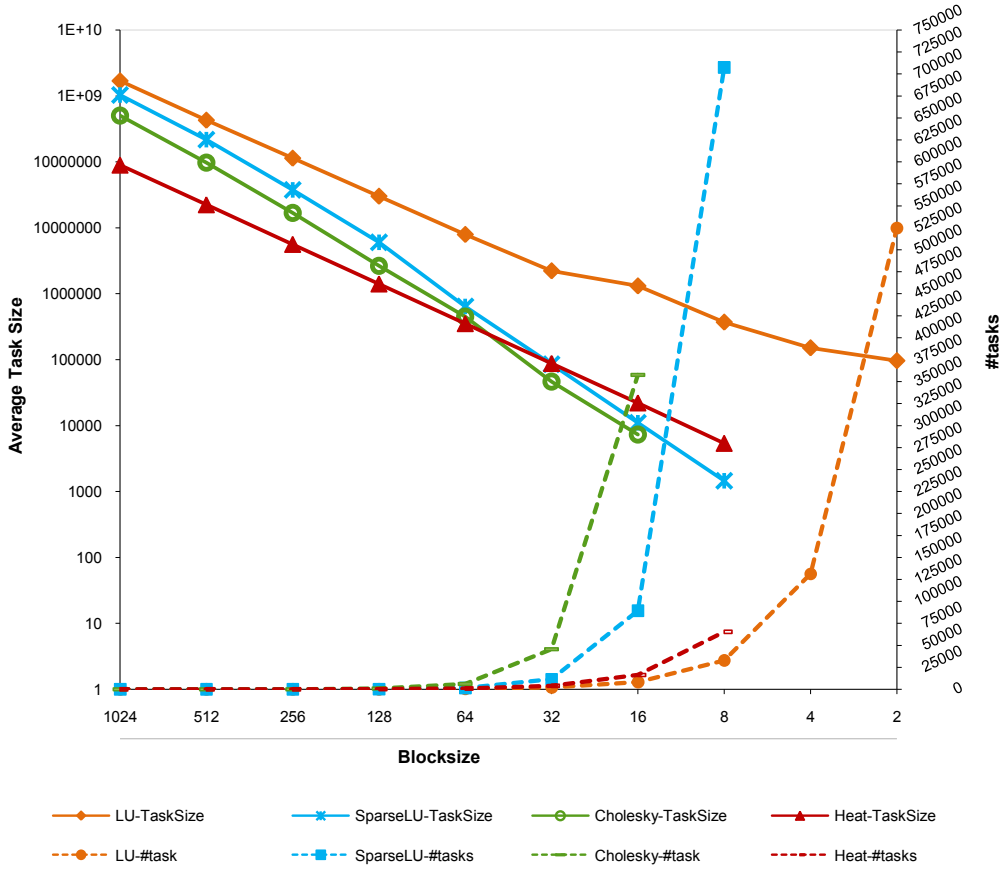


Figure 5.16: Number of tasks and average task size in cycles of Cholesky 2048, SparseLU 2048, Heat 2048 and LU 2048 as function of the block size

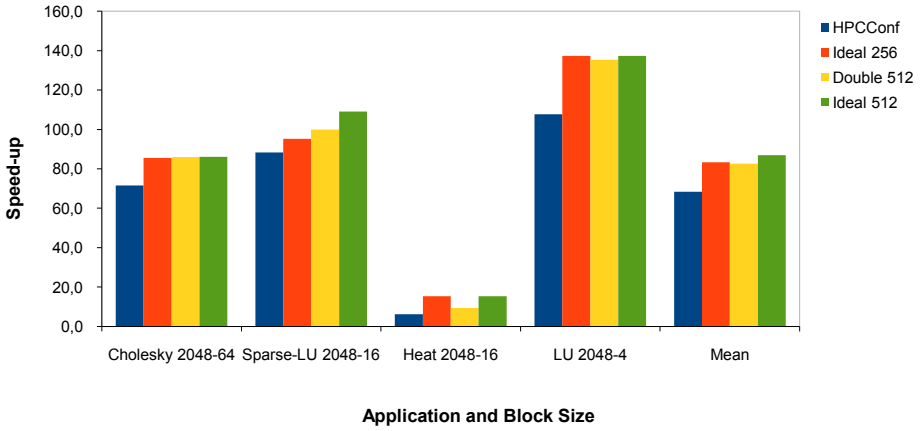


Figure 5.17: Speed-ups obtained for different parallelizations of the Cholesky, SparseLU, Heat and LU applications with different HTSS configurations. First number is the problem size and the second number is the block size

5.5 Summary

In this chapter, the software simulator and the methodology for hardware design space exploration using the simulator have been described. Different HTSS configurations have been analyzed with different amount of hardware resources to provide high performance and different HTSS proposals are presented for different system sizes. Furthermore, HTSS has been compared to the real runtime library of OmpSs, Nanos++.

Chapter 6

Estimation of the Hardware Resources Usage of HTSS

In this chapter, the synthesis results of individual modules of HTSS are presented and discussed. These results are used for estimating the hardware resource usage of different configurations of HTSS. In the first section of this chapter, the experimental framework, synthesis tools and devices that are used through this chapter are introduced. Then, the synthesis results of the individual modules of the hardware prototypes are presented. Subsequently, the estimation and analysis of the hardware resource usage of the integrated prototype, using the configurations resulted from the design exploration are presented comparing it to the initial design.

6.1 Methodology and Experimental Setup

In order to synthesize the modules of HTSS, the Xilinx suite tool has been used. Xilinx integrated software environment (ISE) provides the HDL and schematic editor, logic synthesizer, fitter, and bit-stream generator software as well as a simulator. For obtaining the results of this chapter, the Xilinx synthesis technology tool (XST) version 14.5, Vivado and PlanAhead tools have been used for synthesizing, doing the analysis of the area and frequency of the prototypes and also mapping them on the target device.

During this chapter, several tables are presented. The data of these tables have been obtained from reports of the synthesis tool, (i.e., XST PlanAhead tool), which are reported after synthesizing each module of the proposed designs. In reporting the results, the same terminologies used by XST are employed. These terminologies are described as follows:

- *Distributed memory style* is one possible way for synthesizing memory modules. Using this style, the memory module is synthesized and mapped onto the lookup tables (LUTs) of the FPGA.

- *BRAM memory style* is a method for synthesizing a memory module onto internal blocks of RAMs of the FPGA. Using this style, the LUTs of the FPGA are preserved for other parts of the design.

- *Slice logic utilization* includes the number of registers and look-up tables (LUTs) slices that are used by the modules. LUTs can be used as logic, as memory, or even as shift registers.

- *Slice logic distribution* presents the number of used pairs of LUTs/Flip Flops. It shows how many of these pairs are fully or partially used. In the case of partial usage, the LUT or Flip Flop of a pair is unused.

- *Specific feature* of an FPGA includes Block RAMs, DSP units, general clock buffer etc., but in the tables, only the number of BRAMs are presented because the modules do not use any DSP units.

- *Macro statistics* include the number of required hardware logic units such as RAMs, registers, comparators, multiplexers, adders/subtractors, tri-state buffers

and XOR gates. These numbers are the same for both distributed and BRAM styles.

6.1.1 Target Devices

A Field Programmable Gate Array (FPGA) consists of an array of programmable logic blocks and interconnections. The functionality of its logic blocks and its interconnections are user programmable. The application that is loaded into the FPGA is composed of variable number of circuits which occupy a group of logic blocks in the FPGA.

The main advantage of reconfigurable computing is its ability to increase performance by using hardware execution, while possessing the flexibility of a software solution. In this work, the FPGAs have been used as a suitable fabric to implement the hardware modules of the HTSS designs and measuring its timing constraints. Although a complete hardware implementation of the full HTSS system is out of the scope of this thesis, and left as a future work, in this chapter the suitably of such implementation in a real FPGA device is analyzed. To this end, four different devices of the Xilinx FPGA family have been selected; two devices from Virtex 7 family and two devices from Zynq family, one of the biggest and one of the smallest device in each family. Table 6.1 presents some information of these devices including the number of slice flip flops (FFs), slice look-up tables (LUTs) and available capacity of the distributed and block RAM memories.

Note that since the selected FPGAs belong to Xilinx FPGA family, they have mostly the same LUT structure. Hence, the reported numbers in the tables of this chapter such as number of slice LUTs, registers, memory units and macro statistics are fairly comparable in all of the selected FPGAs.

Table 6.1: Device Information of the target FPGAs

Device	# Slice look-up tables (LUTs)	# LUTs as memory	# Slice flip flops (FFs)	Max distributed RAM (Kb)	Max block RAM (KByte)	Family
Device 1: xc7z020	53200	17400	106400	1088	560	Zynq
Device 2: xc7vh290t	218800	70800	437600	4425	2115	Virtex
Device 3: xc7z100	277400	108200	554800	6762	3020	Zynq
Device 4: xc7v2000t	1221600	344800	2443200	21550	5814	Virtex

6.2 HTSS Modules Synthesis Results

In this section, the synthesis results obtained for each module of the HTSS designs are presented. The goal is to analyze the hardware resources utilized by each module individually, and compare the base and final designs of HTSS[1]. Then, in following sections, the estimation of the hardware resource usage of the different HTSS configurations integrated by these modules are presented and discussed.

6.2.1 Memory Modules

The memory modules have been a key design point of HTSS. Although they are not critical from the reliance point of view, a wrong approach would result in low performance (due to increased access times) and probably would lead to a lot of wasted resources in form of redundant data. As it is explained in Section 4.1, HTSS has three types of memory modules (i.e., TM, DM and VM) for storing meta-data of tasks, dependences of tasks and their subsequent versions. In Table 4.1, in Chapter 4, the characteristics of the memories of the base HTSS design were presented. Note that the capacities of the memory units presented in that table were selected only for VHDL coding of the modules of HTSS prototypes (i.e., HTSS.1, HTSS.2 and HTSS.3). Therefore, they were selected according to the preliminary fields of the memory entries.

After the design space exploration, the proper capacities of the memories have been determined. With the new selected memory entry sizes and the data obtained from the design space exploration, all the memories in the HPCConf system would have a total amount of: 60 Kbytes for the TM, 22 KBytes for the DM and 12 KBytes for the VM, distributed in 12 small memories (i.e., four TMs, four VMs and four DMs) able to manage up to 1024 in-flight tasks. This size could be further reduced by optimizing the TM that is the largest memory in the HPCConf system.

Table 6.2 shows the capacity of memories of the HPCConf configuration found in the previous chapter and the information stored in each entry with its size in bits, for every memory of the final design of HTSS. Note that each memory module entry in this table is aligned to eight-bit word size (i.e., memory data bus is a multiplicand of eight), and there is a power of two number of memory entries. The number of entries for

[1] In this chapter, the base HTSS design means the preliminary design of hardware Task Superscalar before design space exploration with preliminary memory capacities, while the final version of HTSS means an HTSS design with real memory capacities that resulted from design space exploration.

the HPCConf were determined in the design space exploration for each of the memory modules: 512 DM entries, 512 VM entries and 256 TM entries.

Table 6.2: Details of the memory modules of the final HTSS design with HPCConf

DM ($22\ KBytes: 88\times2^{11}$)		**VM ($12\ KBytes: 48\times2^{11}$)**		**TM ($60\ KBytes: 60\times2^{13}$)**			
Capacity per module: 5.5 Kbytes # entries: 512 Entry size: 88 bits # modules: 4 Total HPCConf: 22 KBytes		Capacity per module: 3 Kbytes # entries: 512 Entry size: 48 bits # modules: 4 Total HPCConf: 12 KBytes		Capacity per module: 15 Kbytes # entries: 256 (1 entry ≅ 6 slots) Entry size: 80 × 6 = 480 bits # modules: 4 Total HPCConf: 60 KBytes			
Field of a DM entry	**Size (bits)**	**Field of a VM entry**	**Size (bits)**	**Field of a TM entry: Slot 1 (for task)**	**Size (bits)**	**Field of a TM entry: Slots 2-6 (for 15 dependences, 3 dependences in each slot)**	**Size (bits)**
valid bit	1	version ready	1	valid bit	1	version_id	9
dependency address	64	DM dependency entry	6	gtask_id	64	eORT_id	2
last version address	9	consumers exist	1	# dependences	4	chain dependency	1
dependency instances	10	last consumer TRS address	14	# not ready dependences	4	TM chain address	8
padding	4	next producer existence	1	in_execution	1	TRS chain	2
		next producer TRS address	14	padding	4	chain dependency	4
		version instances	10				
		valid bit	1				

As Table 6.2 shows, every entry in the TM uses six slots: one to manage task information (slot 1 in Table 6.2) and the other five to store information of a maximum of 15 dependencies. As the size of the information of the dependence is only 25 bits, three dependences can fit in one TM slot, and only five slots are needed. Those slots are statically assigned. However, as most tasks have only a small number of dependencies, this memory can be easily optimized by making a dynamic assignment of slots 2 to 6. This technique can lead to reduce the memory size to only 20 KBytes with nearly the same results, as it would be able to store up to 1024 in-flight tasks with up to three dependencies each, or 341 tasks with 15 dependencies each. With this optimization, the total amount of memory in all the modules in the system would be only 54 KBytes.

The memory configuration of the HPCConf of the final HTSS version, resulting of the design space exploration, reduces around 6× (83.2%) the amount of memory necessary of the base designs. That can be observed comparing Table 6.2 and Table 6.3.

The memory modules of the base and final version of HTSS have been implemented in both *distributed* and *BRAM* styles in order to have a perspective of the HTSS designs in both styles. In addition, using this information, the final version of HTSS can be compared to the base HTSS design to highlight the significant improvements applied to the base design.

Tables 6.3 and 6.4 depict the synthesis results of the memory modules of the base HTSS and the final HTSS design for the HPCConf configuration, respectively. As the tables show, the number of LUTs depends on the capacity of the memories, but the number (and size) of registers depends on both capacity and structure of the memory modules. For instance, the number of registers and control sets of DM is much more larger than other memories since the way associatively has to be implemented. In BRAM style, the whole structure of TM and DM is embedded in the blocks of RAMs. DM includes eight separate RAMs to be implemented as an eight-way set-associative memory (see Figure 4.3) which are embedded in the blocks of RAM. The rest parts of DM are mapped onto the LUTs of target FPGA. Synthesis reports of the memory modules also show that all the memory modules are implemented with only RAMs and registers but DM, that also needs several comparators and multiplexers. The reason is the complex structure of DM, as an eight-way set associative memory.

As it was mentioned previously, the design of the memory modules has been done in a way that accessing to the memory units takes two pipelined cycles: one cycle for setting the control signals (enabling access mode) and the other for accomplishing the operation (e.g., writing or reading). The only exception here is in the DM. Reading from a way of DM takes three cycles. The memory modules have synchronous *read* and *write* enables. In all cases, the cycle of setting the control signals has been overlapped with the other operations of the FSMs.

Table 6.3: Synthesis results of the memory modules of the base HTSS design

			Task Memory (TM)	Version Memory (VM)	Dependency Memory (DM)
Capacity		Capacity per module	200 KBytes	160 KBytes	200 KBytes
		# modules	2	1	1
		Total HTSS.3	400 KBytes	160 KBytes	200 KBytes
Distributed Style	**Slice Logic Utilization**	# Slice Registers	200	160	12321
		# Slice LUTs	27633	22112	25264
		# LUTs as Logic	2033	1632	6576
		# LUTs as Memory	25600	20480	18688
	Slice Logic Distribution	# LUT / FlipFlop (L/FF) pairs	27633	22112	33960
		#L/FFwith unused Flip Flop	27433	21952	21639
		#L/FFwith unused LUT	0	0	8696
		# fully used LUT-FF pairs	200	160	3625
		# unique control sets	2	1	722
BRAM Style		# Block RAM	50	40	36

Table 6.4: Synthesis results of the memory modules of the final HTSS design (i.e., HPC-Conf)

			Task Memory (TM)	Version Memory (VM)	Dependency Memory (DM)
Capacity			60KByte	12KByte	22KByte
Distributed Style	**Slice Logic Utilization**	# Slice Registers	80	48	1370
		# Slice LUTs	11072	1640	2808
		# LUTs as Logic	832	104	732
		# LUTs as Memory	10240	1536	2077
	Slice Logic Distribution	# LUT / FlipFlop (L/FF) pairs	11072	1640	3773
		#L/FFwith unused Flip Flop	10992	1592	2404
		#L/FFwith unused LUT	0	0	966
		# fully used LUT-FF pairs	80	48	403
		# unique control sets	1	1	80
BRAM Style		# Block RAM	20	3	4

6.2.2 Main Modules

In this section, the synthesis results of main modules (with their memory included) of the HTSS prototypes and the final version of the HPCConf of HTSS are presented.

Table 6.5 details the results obtained when synthesizing the main modules with distributed memory modules, while Table 6.6 presents the synthesis results of the main modules when all possible memory modules are embedded into blocks of RAMs. In the distributed style, the whole modules, including memory and logic units are implemented using LUTs, while in the BRAM style, the whole storage of the memory unit(s) of the modules are embedded into the blocks of RAMs and the rest of the modules are implemented using LUTs. In these tables, the frequency of the clock signals, utilization and distribution of the slice logic units and number of BRAM memories used are presented. The results presented in those tables are for individual modules. For each type of module there may be some differences depending on the version of the system prototype: HTSS.1, HTSS.2, HTSS.3, HPCConf and MinConf. Note that the first three prototypes are improvements made on the original design to reduce the latency of the operations, and the last two are basically the HTSS.3 with a significant reduction of the system memory requirements; done after theă design space exploration. Those prototypes were explained in the previous chapters.

Due to the big FSM and large memory storage of TRS and iTRS, these modules operate at the lowest frequency compared to the others. GW and iGW work with the highest frequency due to their small FSM that does not access to any memory unit. As the tables show the final version of the HTSS (HPCConf and MinConf) have smaller modules compared to the base versions (HTSS.1, HTSS.2 and HTSS.3) due to the less amount of memory used.

Tables 6.5 and 6.6 show that the modules with more complex functionality and more memory storage occupy more registers and LUTs (e.g., TRS and eORT). Since GW is only responsible for issuing tasks and dependences to the pipeline, it has only two simple FSMs and therefore, it uses less LUTs than the others. The tables also show that the improvements applied to GW and TRS cause the new modules (i.e., iGW and iTRS) to utilize less LUTs and registers. Furthermore, LUTs and registers of eORT are less than the sum of the LUTs and registers of both OVT and ORT. The number of control

Table 6.5: Hardware resource usage of the main modules of the HTSS designs with distributed RAMs

Component		GW	iGW	TRS	iTRS	iTRS	OVT	ORT	eORT	eORT
Prototype(s)		HTSS.1 & HTSS.2	HTSS.3 & HPCConf & MinConf	HTSS.1 & HTSS.2	HTSS.3	HPCConf & MinConf	HTSS.1	HTSS.1	HTSS.2 & HTSS.3	HPCConf & MinConf
Frequency (MHz)		402.60	405.40	214.78	219.79	219.79	348.03	252.16	252.16	252.16
Slice Logic Utilization	# Slice Registers	3489	2490	4120	3868	2114	4344	14144	17320	14512
	# Slice LUTs	2736	1959	27953	26112	14054	27204	29044	55494	20073
	# LUTs as Logic	2736	1959	15153	13312	12760	5178	10356	16326	15622
	# LUTs as Memory	0	0	12800	12800	1294	20475	18688	39168	4451
Slice Logic Distribution	# LUT / FlipFlop (L/FF)	4153	3062	30018	28119	15980	29742	37468	60062	27450
	#L/FF with unused FF	664	572	26563	23716	13391	25400	23324	48376	21036
	#L/FF with unused LUT	1417	1103	1969	2639	1552	2538	8424	3210	2480
	# fully used LUT-FF	2072	1387	1486	1764	1037	1806	5720	8476	3934
	# unique control sets	2012	1077	5524	4261	4114	3800	5006	7120	6080

Table 6.6: Hardware resource usage of the main modules of the HTSS designs with block RAMs

Component		GW	iGW	TRS	iTRS	iTRS	OVT	ORT	eORT	eORT
Prototype(s)		HTSS.1 & HTSS.2	HTSS.3 & HPCConf & MinConf	HTSS.1 & HTSS.2	HTSS.3	HPCConf & MinConf	HTSS.1	HTSS.1	HTSS.2 & HTSS.3	HPCConf & MinConf
Frequency (MHz)		402.60	405.40	214.78	219.79	219.79	348.03	252.16	252.16	252.16
Slice Logic Utilization	# Slice Registers	3489	2490	6428	6100	4870	4186	14867	17159	13026
	# Slice LUTs	2736	1959	12880	12650	724	5096	10003	13376	764
	# LUTs as Logic	2736	1959	12880	12650	724	5096	10003	13376	764
	# LUTs as Memory	0	0	0	0	0	0	0	0	0
Slice Logic Distribution	# LUT / FlipFlop (L/FF)	4153	3062	17010	16284	9537	7515	19877	24745	11320
	#L/FFwith unused FF	664	572	10736	10334	6113	3329	5010	7586	3144
	#L/FFwith unused LUT	1417	1103	3624	3470	1880	2419	9874	11369	5698
	# fully used LUT-FF	2072	1387	2650	2480	1544	1767	4993	5790	2478
	# unique control sets	2012	1077	5440	5111	4990	3804	5602	8066	3210
Specific Feature	# Block RAM	0	0	26	26	5	41	36	76	2

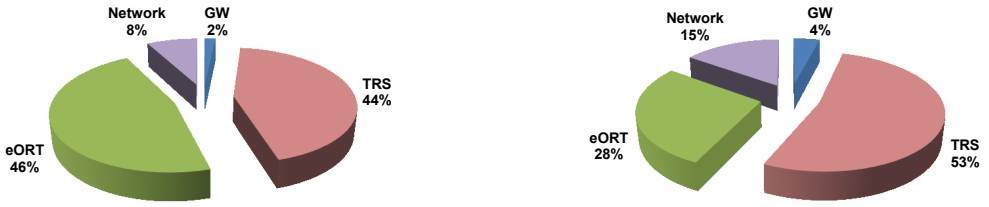
sets is related to the complexity of the modules; the more complex structure, the more number of control sets used.

6.3 Evaluation and Analysis of the HTSS Design

In this section, the synthesis results of the HTSS modules are analyzed and evaluated. Then, based on these results, the hardware usage of HTSS.3, HPCConf and MinConf is estimated and discussed. HTSS.3 consists of one iGW, one eORT, two iTRSs and one TS. As it is described in the previous section, HPCConf is an HTSS configuration suitable for high performance computing (many-core systems) with the minimum resources that provide high performance. From the design space exploration performed in Chapter 5 it has been obtained that HPCConf consists of one iGW, four eORTs, four iTRSs and one TS. MinConf is an HTSS configuration suitable for small multi-core systems with one iGW, one eORT, one iTRS and one TS. The goal is to compare the estimated hardware usage of the base HTSS design based on the work of Etsion et al. [6] and the final design of HTSS proposed in this work. Furthermore, in order to study the feasibility of the implementation of the hardware designs, the size of the hardware designs are evaluated in order to be mapped on the different selected devices explained in Section 6.1.1.

The pie charts of Figures 6.1 and 6.2 are graphical representations of LUTs utilization of the HTSS.3 and HPCConf, respectively, when all the memory modules are synthesized in distributed style and in BRAM style. The percentage shown in the figures are related to the total number of LUTs used in each design. Note that in BRAM memory style, the number of LUTs used by the modules excludes the memories, because the memories are embedded in blocks of RAMs. The figures also show the necessary hardware to implement their internal network connection. Table 6.7 shows for each design the number of queues and arbiters used in its implementation. As it can be seen, the network requirements of the prototypes grow with the number of modules. However, this is not a problem since the current network is more-than-enough to support a many-core system and it is implementable in the current devices.

Considering the synthesis results of integrated HTSS prototypes, the possibility of implementing the hardware prototypes on the selected FPGAs has been studied. Figure 6.3 illustrates the percentage of the LUTs of the four selected devices that are used by HTSS.3, HPCConf and MinConf prototypes. As it can be seen in Figure 6.3 using BRAMs to implement memories, any of the selected devices can be used to host the prototypes taking into account only LUTs utilization.



(a) (b)

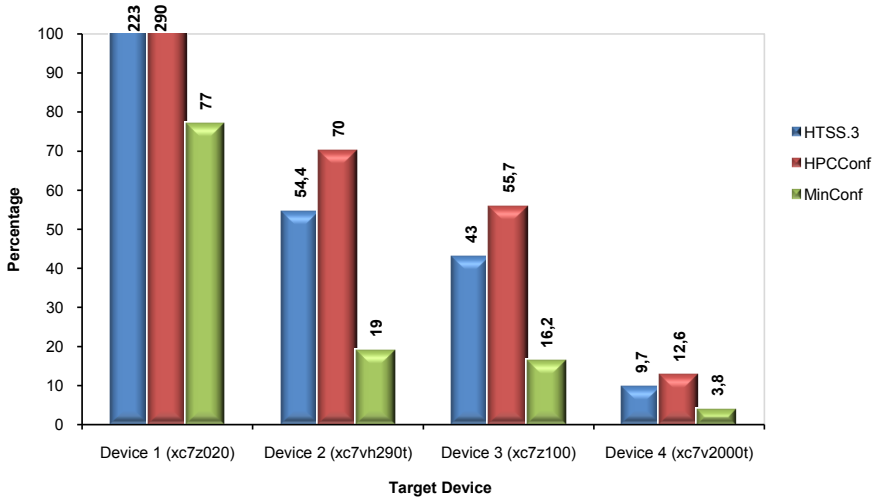
Figure 6.1: LUTs usage of the modules of HTSS.3 (a) with distributed memories, (b) with BRAM memories

(a) (b)

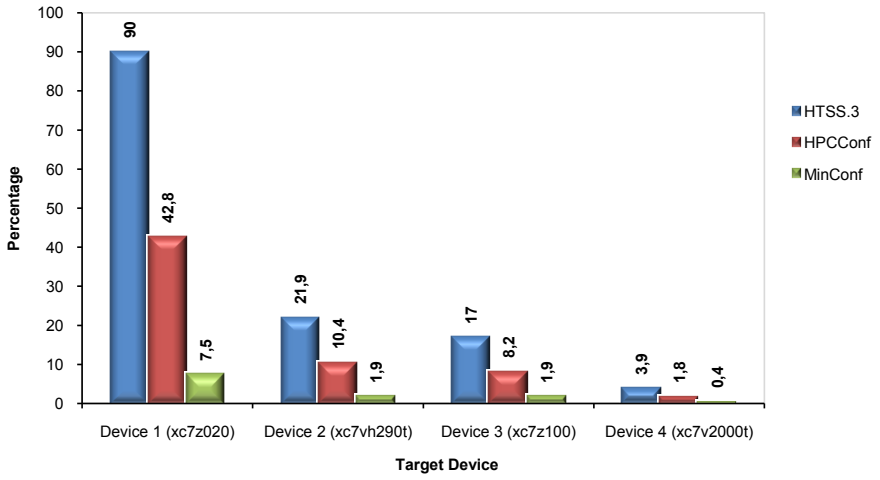
Figure 6.2: LUTs usage of the modules of HPCConf (a) with distributed memories, (b) with BRAM memories

Table 6.7: Number of FIFOs and arbiters of each HTSS prototypes

HTSS design	Number of FIFOs	Number of arbiters
HTSS.1	23	6
HTSS.2	21	5
HTSS.3	17	4
HPCConf	42	8
MinConf	7	1



(a)



(b)

Figure 6.3: Percentage of LUTs of the devices used by the HTSS.3, HPCConf and MinConf (a) with distributed RAMs, (b) with BRAMs

The diagrams of Figure 6.3 shows that using BRAMs all the prototypes can be placed on the FPGAs under the point of view of LUTs utilization. However, in this case, the available memory resources become the more restrictive resource on the devices. This especially happens for the HTSS.3 design because in this design most of the hardware resources are dedicated to memory units. On the other hand, flip flops (FF) never become a limit to implement the designs in the selected devices.

Figure 6.4 presents percentage of the memory resources of the devices used by the prototype. Figure 6.4-a shows the percentage of the memory LUTs of the devices that are used by the distributed memories of the prototypes, while Figure 6.4-b presents the percentage of the block RAMs of each device that are utilized by the memory modules of the prototype.

Table 6.8 summarizes the test results of Figures 6.3 and 6.4. This table shows the target FPGA where the HTSS designs can be mapped. HTSS.3 can be placed on *Device 2* and *Device 4* with distributed memory style, while with BRAM memory can be placed on all the selected devices except *Device 1*. HPCConf can be mapped on all the devices in both memory styles, but *Device 1*. In this case, HPCConf using distributed memory style is larger than this device. MinConf is small enough to be placed on all the selected devices.

Table 6.8: Capacity test of mapping the HTSS design on the selected devices

	Distributed memory style			BRAM memory style		
	HTSS.3	HPCConf	MinConf	HTSS.3	HPCConf	MinConf
Device 1 (xc7z020)	✗	✗	✓	✓	✓	✓
Device 2 (xc7vh290t)	✓	✓	✓	✓	✓	✓
Device 3 (xc7z100)	✗	✓	✓	✓	✓	✓
Device 4 (xc7v2000t)	✓	✓	✓	✓	✓	✓

In order to give an approximation of the number of times that the memories of the designs can be increased, Table 6.9 shows the number of HTSS that can be mapped in a target device for each of the designs. Although Chapter 5 has demonstrated that the selected memory is enough to deal with real application, having more memory would result in the capacity of managing more in-flight tasks and, consequently, the capacity to discover more parallelism ahead of the current execution point. The numbers in Table

6.9 also show the significant improvement that the hardware designs have undergone. Indeed, HPCConf is able to manage the same number of tasks than HTSS.3 with better throughput (due to its larger number of modules) using considerably less resources.

Table 6.9: Number of HTSS designs that could be mapped on the selected devices

	Distributed memory style			BRAM memory style		
	HTSS.3	**HPCConf**	**MinConf**	**HTSS.3**	**HPCConf**	**MinConf**
Device 1 (xc7z020)	0	0	1	1	2	13
Device 2 (xc7vh290t)	1	1	5	3	9	52
Device 3 (xc7z100)	0	1	6	3	12	52
Device 4 (xc7v2000t)	4	7	26	10	52	250

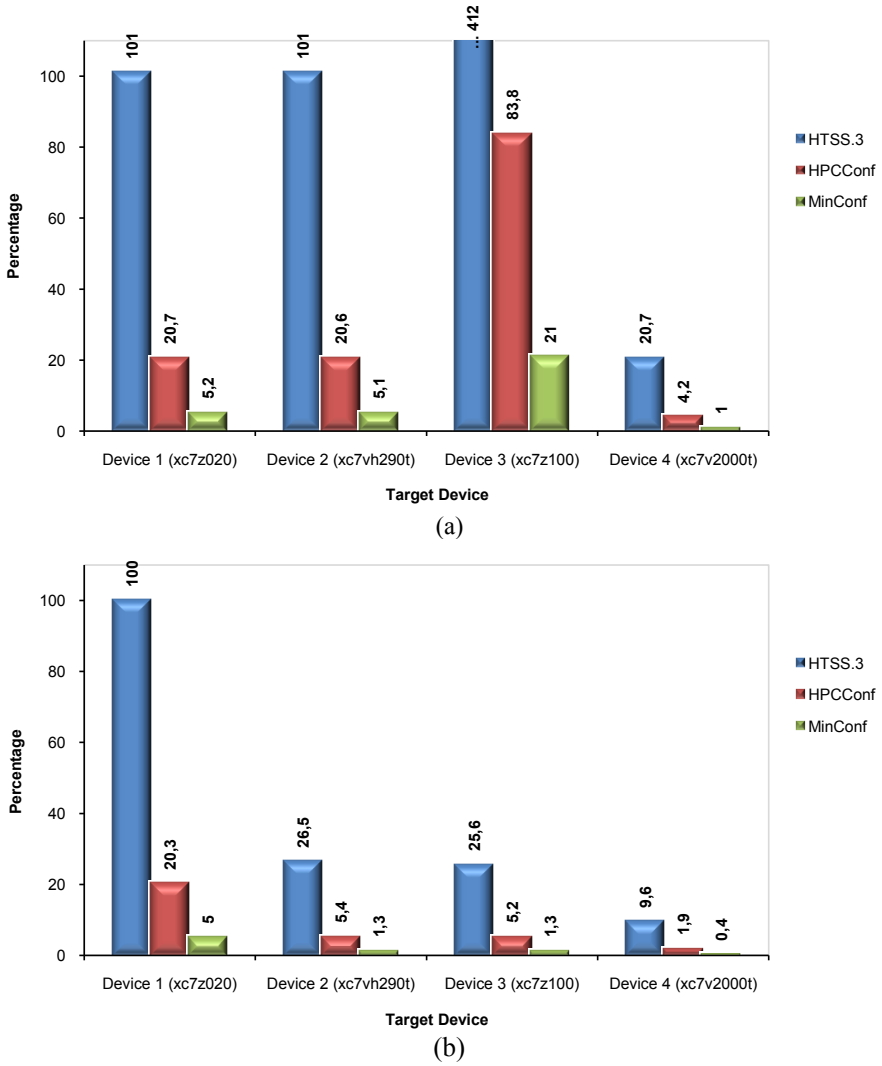


Figure 6.4: Percentage of memory logics of the devices used by the prototypes, (a) with distributed RAMs, (b) with BRAMs

6.4 Summary

This chapter presented the synthesis results, estimation and evaluation of the hardware resources usage of the HTSS designs. After introducing the experimental setup and the devices selected for this study, the synthesis results of all the individual components of the HTSS prototypes are presented. Based on these results, the hardware usage of the different HTSS configurations resulted from the design space exploration are estimated.

Those results show that an HPCConf is implementable in current devices, with the possibility of increasing its capacity to support more in-flight tasks, and then, to exploit more application parallelism.

Chapter 7

Conclusions and Future Work

The goal of this thesis has been to propose a realistic hardware design of the Task Superscalar architecture, a dataflow task scheduler created to accelerate the execution of applications annotated with the OmpSs programming model.

To this end, this document first presents the thesis objectives and goals. After that, a brief review of the von-Neumann and dataflow computing models is presented explaining how this two models can be combined into new hybrid architectures. Also the strengths and flaws of these architectures are explained along with the properties of the new programming models that they support. It has been stated how these new programming models increasingly rely on software runtime models to alleviate the parallel programming burden and has been shown how the software inefficiencies lead to limited performance gains when the parallelism explodes.

Once the potential benefits of a hardware task manager have been discussed, the Task Superscalar architecture design has been presented as a first attempt to implement such a system. After that, a new and more realistic hardware implementation has been designed and presented in this document along with two improved designs that significantly reduce processing time and hardware resource usage over the initial proposal. With these new designs, a hardware implementation has been coded in VHDL in order to obtain real data of the time constraints of such systems.

Based on the VHDL implementation of each of the modules a cycle accurate software simulator has been developed in order to further improve the designs by detecting and correcting the system bottlenecks and perform a full design space exploration of a real and full hardware prototype.

As a result of this work, a realistic and implementable hardware prototype consisting of 10 modules (one gateway, four dependency chain trackers, four task reservation stations and one scheduler) has been proposed. The proposed design has demonstrated its ability to deal with systems that manage up to 256 processors with a set of real benchmarks. The results show that the resulting system can obtain speed-ups closer to the maximum for the parallelization strategy of the analyzed applications (up to 100x in the tests) using a reasonable amount of additional hardware memory (less than 100 KB).

The prototype has also been compared to its runtime software alternative (Nanos++) and the significant performance benefits of a hardware implementation have been demonstrated. The results showed that the hardware approach is much more efficient than the

software alternative for the whole set of benchmark applications when using fine-grained parallelization strategies being less sensitive to the granularity of the tasks.

Finally, the hardware resource usage of the final prototype has been estimated showing the significant reduction achieved over the original proposal. The number of necessary modules and their interconnection network has been significantly reduced, from 14 modules to 10, along with the sizes of their internal memories, from around 7MB to less than 100KB.

7.1 Future Work

Although this thesis has demonstrated the effectiveness of an implementable hardware task manager to deal with the challenges of exploiting next-generation many-core systems, several works remain to be done to widen the applicability of the system.

To fully support the programming model, new types of dependencies that allow concurrent execution and reductions (already supported in the software runtime) should be introduced in the hardware system. In addition, the hardware should be connected to its software counterpart in order to achieve a full working proof-of-concept system. Both the hardware and the software side could also be extended to include support to nested tasks scheduling which is a natural evolution of the programming model.

Finally, in the hardware side, global dependence management between all tasks (and not only between sibling tasks) can be introduced and tested in order to compare it to the current implementation in software that only supports local dependences (i.e. between sibling tasks). Indeed, this implementation can lead to propose an extension to the actual programming model.

Appendices

This thesis dissertation presents four appendices including supplementary tables of the main text. The information of these tables has been used for describing the VHDL designs. In appendix A, common fields which are used in the VHDL description of the prototypes are introduced. Appendix B presents the fields of the entries of the memory modules of the designs. Details of the communication packets are shown in Appendix C. Finally, in Appendix D, supplementary tables of results of design space exploration using SimTSS are presented.

Appendix A

Definition of Common Fields

In this appendix, the definition of common fields used in the VHDL description of the hardware Task Superscalar architecture memory entries and packets are presented. Table A.1 shows for each field, its name, its original size in bits and a brief definition of the information that it stores.

Table A.1: Common fields in the hardware implementation of Task Superscalar architecture

Field Name	Size (bits)	Definition
ANI	14	Absolute number of instances (users) for a dependence
dir	2	Direction of each dependence (Output = 00, InputOut = 01, Input = 10, Direct = 11)
DM_adrs	14	Address of the DM entries including <index_adrs,way_index>
gtask_id	64	General task identifier that indicates address of each task in the main memory
HMS	1	Have more space in TM?
i_cnt	4	Counter of input dependences in a task
max_num_depen	4	Maximum number of dependences which is equal to 15
num_depen	4	Number of dependences
num_version	8	Number of versions for each dependence
operand_adrs	64	Address of each operand in the main memory
OVT_num_users	14	Number of users for each version of a dependence
p_cnt	4	Counter of pending dependences (i.e., not ready) in a task
packet_id	4	Identifier for selecting up to 16 different packets
depen_id	32	Identifier for each dependence including < TRS_id, slot_id, depen_index >
depen_idx	4	Index of a dependence in a task
slot_id	13	Address for each task in a TM (The four lowest bits of each slot_id are zeros).
task_id	17	Identifier for each task including < TRS_id, slot_id >
TRS_id	4	Identifier for selecting up to 16 TRSs
version_id	16	Address of the versions in the OVT memory (3 highest bits are zeros).

Appendix B

Structures of Memory Modules

This appendix presents the structures of the memory modules of the hardware designs of the Task Superscalar architecture. HTSS designs have three types of memory modules (i.e., DM, VM and TM) embedded in the components of the front-end pipeline of the Task Superscalar architecture. Tables B.1, B.3 and B.2 presents the fields of each entry of these memories. Table B.1 shows the fields of each entry of DM which is organized as an eight-way set associative memory. TM and VM are both indexed memories. TM is divided into blocks. In the initial design each block had 16 slots, one for meta-data of tasks which is detailed in Table B.3, and the others used to save meta-data of up to 15 dependences. In the final design, only 6 slots per task were used. The first slot saved the meta-data of the task and the other 5 slots saved each the meta-data of up to three dependences.

Table B.1: Details of the dependence memory (DM)

The Base Design			**The Final Design**	
Field	**Size (bits)**		**Field**	**Size (bits)**
valid bit	1		valid Bit	1
dir	2		dep. addr.	64
depen_id <TRS_id, slot_id, depen_idx>	21		last ver. addr.	9
version_id	13		dep. instances	10
prev_version_id	13		padding	4
ANI	14			
#version	8			
gtask_id	64			
operand_size	8			
Total size of one entry	**144**		**Total size of one entry**	88

Table B.2: Details of the version memory (VM)

The Base Design			The Final Design	
Field	**Size (bits)**		**Field**	**Size (bits)**
valid bit	1		version ready	1
version_ready	1		DM dep. entry	6
Dir	2		consumers exist	1
OVT_nusers	14		last consumer TRS addr.	14
version_id of next version	13		next producer exists	1
depen_id of TCS (Top of Consumers Stack)	21		next producer TRS addr.	14
operand_adrs	64		version instances	10
operand_size	8		padding	1
ORT_adrs	14			
depen_id <TRS_id, slot_id, depen_idx>	21			
padding	1			
Total size of one entry	**160**		**Total size of one entry**	**48**

Table B.3: Details of the task memory (TM)

The Base Design			**The Final Design**	
Slot 1 (for task)			**Slot 1 (for task)**	
Field	**Size (bits)**		**Field**	**Size (bits)**
valid bit	1		valid bit	1
gtask_id	64		gtask_id	64
#dependency	4		#dependency	4
P_cnt	4		# not ready dependency	4
I_cnt	4		in_execution	1
code_adrs	64		padding	6
code_size	16			
padding	3			
Total size of one entry	**160**		**Total size of one entry**	**80**
Slots 2-16 (for 15 dependences) One slot = One dependence			**Slots 2-6 (for 15 dependences) One slot = Three dependences**	
Field	**Size (bits)**		**Field**	**Size (bits)**
valid bit	1		version_id	9
Ready	1		eORT_id	2
dir	2		chain dependency	1
depen_id <TRS_id, slot_id, depen_idx>	21		TM chain address	8
operand_adrs	64		TRS chain	2
operand_size	8		chain dependency	4
rename_id	13		padding (per slot)	2
prev_depen_id <TRS_id, slot_id,depen_idx>	21			
prev_gtask_id	64			
padding	5			
			Total size of one dependence	**26**
Total size of one entry	**200**		**Total slot size**	**80**

Appendix C

Definition and Format of Packets

This appendix presents the details of the packets that are used in the VHDL implementation of the prototypes of HTSS. The definition of the packets are described in Table C.1. Tables C.2 to C.13 details the fields of the packets with their size. For each packet, the corresponding table presents its fields in the base design and the ones used in the final design that significantly reduce traffic.

Table C.1: Information of packets

Packet name	Function	Origin module	Dest. module	Details in table
ContIssue	It notifies the GW that a task is terminated and its place in the Task memory becomes free.	TRS	GW	C.2
CreateVersion	It is used to create a version and/or to update the existing entry depending on the direction and existence of previous version(s).	ORT	OVT	C.3
DataReady	It notifies another task that a task is finished and its output dependence is ready.	TRS/ OVT	TRS	C.4
DepenORT (DepeneORT)	It is used for sending indirect dependence from the GW to the ORT (or eORT) for data dependency analysis.	GW	ORT/ eORT	C.5
DepenTRS	It is used for sending a dependence.	OVT/ eORT	TRS	C.6
DirectDepen	It is used for sending a direct (scalar) dependence.	GW	TRS	C.7
DropDepen	It is sent for informing that a dependence of a finished task is released.	TRS	OVT	C.8
DropVersion	It is used to get permission for releasing a version of a dependence.	OVT	ORT	C.9
Execute	It includes the meta-data of a ready task for sending to one of the execution units.	TRS	TS	C.10
Finish	It notifies that execution of the task has been finished.	TS	TRS	C.11
Issue	It includes meta-data of a task and is sent for allocating a slot of Task memory.	GW	TRS	C.12
IssueAck	It is an acknowledge message from a TRS to the GW. This packet says the allocated address in Task memory, and whether the TRS has more space for next task allocation or not.	TRS	GW	C.13

Table C.2: `ContIssue` packet

Field	Size (bits)	The base version	The final version
valid bit	1	√	√
packet_id	4	√	✗
task_id <TRS_id, slot_id>	17	√	√(10 bits)
Total size		**22 bits**	**11 bits**

Table C.3: `CreateVersion` packet

Field	Size (bits)	The base version	The final version
valid bit	1	√	This packet does not exist in the final version
packet_id	4	√	
first_time	1	√	
dir	2	√	
version_id	13	√	
operand_adrs	64	√	
operand_size	8	√	
depen_id<TRS_id, slot_id, depen_idx>	21	√	
prev_ depen_id<TRS_id, slot_id, depen_idx>	21	√	
prev_version_id	13	√	
num_users	14	√	
DM_adrs	14	√	
prev_gtask_id	64	√	
Total size		**240 bits**	

Table C.4: `DataReady` packet

Field	Size (bits)	The base version	The final version
valid bit	1	√	√
packet_id	4	√	√(1 bit)
dir	2	√	X
src_depen_id <TRS_id, slot_id, depen_idx>	21	√	X
des_depen_id <TRS_id, slot_id, depen_idx>	21	√	√(14 bits)
version_id	13	√	X
operand_adrs	64	√	X
Total size		**126 bits**	**16 bits**

Table C.5: `DepenORT` (`DepeneORT`) packet

Field	Size (bits)	The base version	The final version
valid bit	1	√	√
packet_id	4	√	✗
ready	1	√	✗
dir	2	√	√
depen_id <TRS_id, slot_id, depen_idx>	21	√	√(14 bits)
operand_adrs	64	√	√
operand_size	8	√	✗
gtask_id	64	√	✗
Total size		**165 bits**	**81 bits**

Table C.6: `DepenTRS` packet

Field	Size (bits)	The base version	The final version
valid bit	1	√	√
packet_id	4	√	√(1 bit)
ready	1	√	√
dir	2	√	✗
depen_id <TRS_id, slot_id, depen_idx>	21	√	√(14 bits)
operand_adrs	64	√	✗
operand_size	8	√	✗
prev-depen_id <TRS_id, slot_id, depen_idx>	21	√	√(14 bits)
prev_gtask_id	64	√	✗
version_id	13	√	√(11 bits)
Total size		**199 bits**	**42 bits**

Table C.7: `DirectDepen` packet

Field	Size (bits)	The base version	The final version
valid bit	1	√	√
packet_id	4	√	✗
ready	1	√	√
dir	2	√	√
depen_id <TRS_id, slot_id, depen_idx>	21	√	√(12 bits)
value	64	√	√
Total size		**93 bits**	**80 bits**

Table C.8: `DropDepen` packet

Field	Size (bits)	The base version	The final version
valid bit	1	√	√
packet_id	4	√	X
version_id	13	√	√ (11 bits)
depen_id <TRS_id, slot_id, depen_idx>	21	√	X
Total size		**39 bits**	**12 bits**

Table C.9: `DropVersion` packet

Field	Size (bits)	The base version	The final version
valid bit	1	√	This packet does not exist in the final version
packet_id	4	√	
DM_adrs <index, way_idx>	14	√	
version_id	13	√	
operand_adrs	64	√	
operand_size	8	√	
OVT_num_users	14	√	
Total size		**118 bits**	

Table C.10: `Execute` packet

Field	Size (bits)	The base version	The final version
valid bit	1	√	√
packet_id	4	√	X
task_id <TRS_id, slot_id>	17	√	√(10 bits)
gtask_id	64	√	X
num_depen	4	√	X
code_address	64	√	√
code_size	16	√	X
Total size		**170 bits**	**75 bits**

Table C.11: `Finish` packet

Field	Size (bits)	The base version	The final version
valid bit	1	√	√
packet_id	4	√	X
task_id<TRS_id,slot_id>	17	√	√(10 bits)
gtask_id	64	√	X
Total size		**86 bits**	**11 bits**

Table C.12: `Issue` packet

Field	Size (bits)	The base version	The final version
valid bit	1	√	√
packet_id	4	√	X
gtask_id	64	√	X
num_depen	4	√	√
TRS_id	4	√	√(2 bits)
code_adrs	64	√	√
code_size	16	√	X
slot_id	13	√	√(8 bits)
Total size		**170 bits**	**79 bits**

Table C.13: `IssueAck` packet

Field	Size (bits)	The base version	The final version
valid bit	1	√	This packet does not exist in the final design.
packet_id	4	√	
HMS	1	√	
task_id<TRS_id, slot_id>	17	√	
Total size		**23 bits**	

Appendix D

Supplementary of SimTSS Results

This appendix section presents some supplementary results obtained in the SimTSS design space exploration.

Table D.1: Number of cycles obtained when executing the SparseLU benchmark with different number of TRSs and different number of task memory entries

SparseLU TM entries	1 TRS	2 TRSs	4 TRSs	8 TRSs	16 TRSs
8	14418358	7547674	4442248	3071120	2404377
16	7552170	4444883	3072492	2404709	2065427
32	4453660	3076403	2405424	2064783	2035172
64	3089870	2412849	2065092	2035586	2035992
128	2439063	2071540	2036288	2037860	2037859
256	2140966	2037440	2039089	2038448	2038933
512	2090604	2039608	2041511	2041237	2041369
1024	2089571	2041191	2041502	2041947	2041848
2048	2089555	2041191	2041502	2041947	2041853
4096	2089555	2041191	2041502	2041947	2041853
8192	2089555	2041191	2041502	2041947	2041853
16384	2089555	2041191	2041502	2041947	2041853

Table D.2: Number of cycles obtained when executing the Heat benchmark with different number of TRSs and different number of task memory entries

Heat TM entries	1 TRS	2 TRSs	4 TRSs	8 TRSs	16 TRSs
8	979751	687627	461517	285533	173700
16	689166	462002	285632	173928	112323
32	463117	286482	174174	112554	105082
64	288082	174753	112946	104924	105017
128	176308	113832	105611	105070	105017
256	136439	106412	105455	105070	105017
512	136439	106341	105455	105070	105017
1024	136439	106341	105455	105070	105017
2048	136439	106341	105455	105070	105017
4096	136439	106341	105455	105070	105017
8192	136439	106341	105455	105070	105017
16384	136439	106341	105455	105070	105017

Table D.3: Number of cycles obtained when executing the LU benchmark with different number of TRSs and different number of task memory entries

LU TM entries	1 TRS	2 TRSs	4 TRSs	8 TRSs	16 TRSs
8	9440107	5376296	3408972	2513362	2349893
16	5450176	3433110	2515597	2351973	2496810
32	3561412	2543162	2358578	2509351	2502957
64	2894149	2397058	2539209	2514779	2501291
128	2748293	2578064	2549252	2515744	2508102
256	2755133	2600195	2549226	2531146	2519169
512	2775087	2631566	2578439	2577064	2527890
1024	2803544	2667078	2625697	2556677	2512166
2048	2819373	2705716	2607917	2532199	2522983
4096	2858827	2647980	2571545	2530849	2523766
8192	2858827	2647980	2571545	2530849	2523766
16384	2858827	2647980	2571545	2530849	2523766

D. SUPPLEMENTARY OF SIMTSS RESULTS

References

[1] P. Bellens, J. Perez, R. Badia, and J. Labarta. **CellSs: A programming model for the Cell BE architecture**. In *Supercomputing (SC)*, New York, NY, USA, 2006. ACM. 2, 34

[2] Perez, R. M. Badia, and J. Labarta. **A dependency-aware task-based programming environment for multi-core architectures**. In *International Conference on Cluster Computing (CC)*, pages 142–151, 2008. 2, 26, 34

[3] P. Bellens, J. M. Perez, F. Cabarcas, A. Ramirez, R. M. Badia, and J. Labarta. **CellSs: Scheduling techniques to better exploit memory hierarchy**. *Scientific Programming*, **17**(1-2):77–95, 2009. 2, 26, 34

[4] M. C. Rinard and M. S. Lam. **The design, implementation, and evaluation of Jade**. *ACM Transaction Programming Language System (TPLS)*, **20**(3):483–545, 1998. 2, 26, 34

[5] J. C. Jenista, Y. Hun Eom, and B. C. Demsky. **OoOJava: Software Out-of-Order execution**. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 57–68, 2011. 2, 26, 34

[6] 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 (HOTPAR)*, 2010. xiii, 2, 31, 32, 35, 39, 44, 52, 53, 59, 123

[7] 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 *International Symposium on Microarchitecture (MICRO)*, pages 89–100, 2010. 2, 19, 21, 31, 35, 53

[8] Fahimeh Yazdanpanah, D. Jimenez-Gonzalez, C. Alvarez-Martinez, and Y. Etsion. **Hybrid dataflow/von-Neumann architectures**. *IEEE Transaction on Parallel and Distributed Systems (TPDS)*, 2013. 2, 9, 19, 31, 52

[9] John von Neumann. **First draft of a report on the EDVAC**. Technical report, United States Army Ordnance Department and the University of Pennsylvania Moore, School of Electrical Engineering, 1945. 11

[10] E. Bloch. **The engineering design of the stretch computer**. In *IRE-AIEE-ACM (Eastern) Computer Conference*, pages 48–58, 1959. 11

[11] T. Agerwala and J. Cocke. **High performance reduced instruction set processors**. Technical report, IBM Technical Journal of Watson Research Center Technical Report RC12434, 1987. 11

[12] Y. N. Patt, W. M. Hwu, and M. Shebanow. **HPS, a new microarchitecture: Rationale and introduction**. In *International Symposium on Microarchitecture (MICRO)*, pages 103–108, 1985. 11, 19

[13] J. A. Fisher. **Very long instruction word architectures and the ELI-512**. *SIGARCH Computing Architecture News*, **11**(3):140–150, 1983. 11

[14] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures (Parallel Computing, Reduced-Instruction-Set, Trace Scheduling, Scientific)*. PhD thesis, Yale University, New Haven, CT, USA, 1985. 11

[15] D. Hillis. *The Connection Machine*. PhD thesis, Department of Electrical Engineering and Computer Science (EECS), MIT, 1988. 12

[16] Thinking Machines Corp. **Connection machine model CM-2 technical ummary**. Technical report, Thinking Machines Corporation Technical Report TR89-1, 1989. 12

[17] J. Nickolls, I. Buck, M. Garland, and K. Skadron. **Scalable parallel programming with CUDA**. *ACM Queue*, **6**(2):40–53, 2008. 12

[18] W. Yamamoto, M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirovsky. **Performance estimation of multistreamed, supersealar processors**. In *Hawaii International Conference on System Sciences*, pages 195–204, 1994. 12

[19] D. M. Tullsen, S. J. Eggers, and H. M. Levy. **Simultaneous multithreading: Maximizing on-chip parallelism**. In *International Symposium on Computer Architecture (ISCA)*, pages 392–403, 1995. 12

[20] J. S. Burton. **Architecture and applications of the HEP multiprocessor computer system.** In *Conference Real time signal processing IV (SPIE)*, pages 241–248, 1981. 12, 17

[21] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. **The Tera computer system**. In *International Symposium on Supercomputing (ICS)*, pages 1–6, 1990. 12, 17

[22] W. D. Weber and A. Gupta. **Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results**. In *International Symposium on Computer Architecture (ISCA)*, pages 273–280. ACM, 1989. 12

[23] K. Kurihara, D. Chaiken, and A. Agarwal. **Latency tolerance through multithreading in large-scale multiprocessors**. In *International Symposium on Computer Architecture (ISCA)*, pages 91–101, 1991. 12

[24] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. **APRIL: A processor architecture for multiprocessing**. In *International Symposium on Computer Architecture (ISCA)*, pages 104–114, 1990. 12

[25] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. **Hyper-threading technology architecture and microarchitecture**. *Intel Technology Journal*, **6**(1):1–12, 2002. 12

[26] J. Clabes, J. Friedrich, M. Sweet, J. DiLullo, S. Chu, D. Plass, J. Dawson, P. Muench, L. Powell, M. Floyd, B. Sinharoy, M. Lee, M. Goulet, J. Wagoner, N. Schwartz, S. Runyon, G. Gorman, P. Restle, R. Kalla, J. McGill, and S. Dodson. **Design and implementation of the POWER5 TM microprocessor**. In *Annual Design Automation Conference (DAC)*, pages 670–672, 2004. 12

[27] Arvind and R. A. Iannucci. **Two fundamental issues in multiprocessing**. In *International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering*, pages 61–88, 1988. 12, 17

[28] R. Buehrer and K. Ekanadham. **Incorporating data flow ideas into von Neumann processors for parallel execution**. *IEEE Transaction Computing*, **36**(12):1515–1522, 1987. 12, 17

[29] M. Herlihy and J. E. B. Moss. **Transactional memory: Architectural support for lock-free data structures**. In *International Symposium on Computer Architecture (ISCA)*, pages 289–300, 1993. 12

[30] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. **Multiscalar processors**. In *International Symposium on Computer Architecture (ISCA)*, pages 414–425, 1995. 12, 19

[31] D. E. Culler, K. E. Schauser, and T. Eicken. **Two fundamental limits on dataflow multiprocessing**. In *IFIP WG 10.3 Conference on Architecture and Compilation Techniques for Medium and Fine Grain Parallelism*, 1993. 12

[32] R. M. Karp and R. E. Miller. **Properties of a model for parallel computations: determinacy, termination, queuing**. *SIAM J. Applied Mathematics*, **14**(5):1390–1411, 1966. 13

[33] J. B. Dennis. **First Version of a Data Flow Procedure Language**. In B. Robinet, editor, *Programming Symposium*, **19** of *Lecture Notes in Computer Science*, pages 362–376. Springer Berlin/Heidelberg, 1974. 13

[34] G. Kahn. **The semantics of a simple language for parallel programming**. In *IFIP Congress*, **74**, 1974. 13

[35] A. L. Davis and R. M. Keller. **Data flow program graphs**. *IEEE Computer*, **15**(2):26–41, 1982. 13

[36] K. M. Kavi, B. P. Buckles, and U. N. Bhat. **A formal definition of data flow graph models**. *IEEE Transaction on Computers (TC)*, **35**(11):940–948, 1986. 13

[37] J. Silc, B. Robic, and T. Ungerer. **Asynchrony in parallel computing: From dataflow to multithreading**. *Journal of Parallel and Distributed Computing (JPDC)*, pages 1–33, 1998. 14, 18, 19, 21

[38] J. B. Dennis and D. P. Misunas. **A preliminary architecture for a basic data-flow processor**. In *International Symposium on Computer Architecture (ISCA)*, pages 126–132, 1975. 14

[39] J. B. Dennis. **Data flow supercomputers**. *IEEE Computer (COMP)*, **13**(11):48–56, 1980. 14

[40] J. B. Dennis. **The Varieties of Data Flow Computers**. In *Advanced Computer Architecture*, pages 51–60. 1986. 14

[41] L. Davis. **The architecture and system method of DDM1: A recursively structured Data Driven Machine**. In *International Symposium on Computer Architecture (ISCA)*, pages 210–215, 1978. 14

[42] A. Plas, D. Comte, O. Gelly, and J.C. Syre. **LAU system architecture: A parallel data-driven processor based on single assignment**. In *International Conference on Parallel Processing (ICPP)*, pages 293–302, 1976. 14

[43] R. Vedder and D. Finn. **The Hughes data flow multiprocessor: Architecture for efficient signal and data processing**. In *International Symposium on Computer Architecture (ISCA)*, pages 324–332, 1985. 14

[44] J. R. Gurd, C. C. Kirkham, and I. Watson. **The Manchester prototype dataflow computer**. *Communication ACM (TPLS)*, **28**(1):34–52, 1985. 15

[45] Arvind and D. E. Culler. **Dataflow architectures**. In *Annual Review of Computer Science vol. 1*, pages 225–253. 1986. 15

[46] Masasuke Kishi, Hiroshi Yasuhara, and Yasusuke Kawamura. **DDDP — a distributed data driven processor**. In *International Symposium on Computer Architecture (ISCA)*, pages 236–242, 1983. 15

[47] N. Ito, M. Sato, E. Kuno, and K. Rokusawa. **The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D**. In *International Symposium on Computer Architecture (ISCA)*, pages 149–156, 1986. 15

[48] D. E. Culler and G. M. Papadopoulos. **The explicit token store**. *Journal of Parallel and Distributed Computing (JPDC)*, **10**(4):289–308, 1990. 15

[49] J. Hicks, D. Chiou, B. S. Ang, and Arvind. **Performance studies of Id on the Monsoon dataflow system**. *Journal of Parallel and Distributed Computing (JPDC)*, **18**(3):273–300, 1993. 15

[50] P. Barahona and J. R. Gurd. **Simulated performance of the Manchester multiring dataflow machine**. In *Parallel Computing*, pages 419–424, 1985. 15

[51] G. M. Papadopoulos and D. E. Culler. **Monsoon: An explicit token-store architecture**. In *International Symposium on Computer Architecture (ISCA)*, pages 82–91, 1990. 15

[52] Arvind, R. S. Nikhil, and K. K. Pingali. **I-structures: Data structures for parallel computing**. *ACM Transaction on Programming Languages and Systems (TOPLAS)*, **11**(4):598–632, 1989. 16

[53] A. H. Veen. **Dataflow machine architecture**. *ACM Computing Surveys (CSUR)*, **18**(4):365–396, 1986. 16

[54] V. P. Srini. **An architectural comparison of dataflow systems**. *IEEE Computer (COMP)*, **19**(3):68–88, 1986. 16

[55] W. A. Najjar, E. A. Lee, and G. R. Gao. **Advances in the dataflow computational model**. *Parallel Computing*, **25**(13-14):1907–1929, 1999. 16, 17

[56] P. C. Treleaven, R. P. Hopkins, and P. W. Rautenbach. **Combining data flow and control flow computing**. *Computer Journal*, pages 207–217, 1982. 17

[57] H. F. Jordan. **Performance measurements on HEP- a pipelined MIMD computer**. In *International Symposium on Computer Architecture (ISCA)*, pages 207–212. ACM, 1983. 17

[58] Joseph E. Requa. **The piecewise data flow architecture control flow and register management**. In *International Symposium on Computer Architecture (ISCA)*, pages 84–89, 1983. 17

[59] G. M. Papadopoulos and K. R. Traub. **Multithreading: A revisionist view of dataflow architectures**. In *International Symposium on Computer Architecture (ISCA)*, pages 342–351, 1991. 17, 21

[60] R. A. Iannucci. **Toward a dataflow/von Neumann hybrid architecture**. In *International Symposium on Computer Architecture (ISCA)*, pages 131–140, 1988. 17

[61] B. Robic, J. Silc, and T. Ungerer. **Beyond dataflow**. *Computing and Information Technology*, **8**(2):89–101, 2000. 18, 19, 21

[62] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V.P. Holmes. **The Epsilon dataflow processor**. In *International Symposium on Computer Architecture (ISCA)*, pages 36–45, 1989. 18

[63] V. G. Grafe and J. E. Hoch. **The EPSILON-2 multiprocessor system**. *Journal of Parallel and Distributed Computing (JPDC)*, **10**(4), 1990. 18

[64] Arvind, L. Bic, and T. Ungerer. **Evolution of Dataflow Computers**. In Jean-Luc Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-Flow Computing*. Prentice Hall, 1991. 18

[65] Y. Kodama, H. Sakane, M. Sato, H. Yamana, S. Sakai, and Y. Yamaguchi. **The EM-X parallel computer: Architecture and basic performance**. In *International Symposium on Computer Architecture (ISCA)*, pages 14–23, 1995. 18

[66] T. Sherwood, S. Sair, and B. Calder. **Predictor-directed stream buffers**. In *International Symposium on Microarchitecture (MICRO)*, pages 42–53, 2000. 18

[67] B. Lee and A. R. Hurson. **Dataflow architectures and multithreading**. *Computer*, **27**(8):27–39, 1994. 18, 19

[68] William Marcus Miller, Walid A. Najjar, and A. P. Wim Bohm. **A quantitative analysis of locality in dataflow programs**. In *International Symposium on Microarchitecture (MICRO)*, pages 12–18, 1991. 18

[69] S. S. Nemawarkar and G. R. Gao. **Measurement and modeling of EARTH-MANNA multithreaded architecture**. In *International Workshop Modeling, Analysis, Simulation of Computing Telecommunication Systems (MASCOTS)*, pages 109–, 1996. 18

[70] X. Tian, S. Nemawarkar, G. R. Gao, H. Hum, O. Maquelin, A. Sodan, and K. Theobald. **Quantitive studies of data-locality sensitivity on the EARTH multithreaded architecture: Preliminary results**. In *International Conference of High-Performance Computers (HIPC)*, pages 362–, 1996. 18

[71] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, and L. J. Hendren. **A study of the EARTH-MANNA multithreaded system**. *Parallel Programming*, **24**(4):319–348, 1996. 18

[72] J. L. Gaudiot, T. DeBoni, J. Feo, W. Bohm, W. Najjar, and P. Miller. **The Sisal model of functional programming and its implementation**. In *International Symposium on Parallel Algorithms/Architecture Synthesis (PAS)*, pages 112–, 1997. 18

[73] K. B. Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, McGill University, Montreal, Quebec, CA, 1999. 18, 21

[74] P. Evripidou and J. L. Gaudiot. **A decoupled graph/computation data-driven architecture with variable-resolution actors**. In *International Conference on Parallel Processing (ICPP)*, pages 405–414, 1990. 18, 19

[75] P. Evripidou and J. L. Gaudiot. **The USC Decoupled Multilevel Dataflow Execution Model**. In Jean-Luc Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-flow Computing*, pages 347–379. Prentice Hall, 1991. 18, 19

[76] W. Bohm, W. Najjar, B. Shankar, and L. Roh. **An evaluation of coarse grain dataflow code generation strategies**. In *Programming Models for Massively Parallel Computers*, pages 63–71, 1993. 18

[77] L. Roh and W.A. Najjar. **Design of storage hierarchy in multithreaded architectures**. In *International Symposium on Microarchitecture (MICRO)*, pages 271–278, 1995. 18, 21

[78] E. Gluck-Hiltrop, M. Ramlow, and U. Schurfeld. **The Stollman dataflow machine**. In *Lecture Notes in Computer Science*, pages 433–457, 1989. 19

[79] E. Zehender and T. Ungerer. **The ASTOR architecture**. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 424–430, 1987. 19

[80] T. Yuba, K. Hiraki, T. Shimada, S. Sekiguchi, and K. Nishida. **The SIGMA-1 dataflow computer**. In *Computer Conference on Exploring Technology: Today and Tomorrow*, pages 578–585, 1987. 19, 21

[81] W. Hwu and Y. N. Patt. **HPSm, a high performance restricted data flow architecture having minimal functionality**. In *International Symposium on Computer Architecture (ISCA)*, pages 297–306, 1986. 19

[82] L. Rauchwerger and D. Padua. **The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization**. In *PLDI*, pages 218–232, 1995. 19

[83] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. **Trace processors**. In *International Symposium on Microarchitecture (MICRO)*, pages 138–148, 1997. 19

[84] J. Silc, B. Robic, and T. Ungerer. *Processor Architecture: From Dataflow to Superscalar and Beyond*. Springer-Verlag, 1999. 19, 21

[85] R. A. Iannucci, G. R. Gao, R. H. Halstead Jr., and B. Smith. *Multithreaded Computer Architecture: A Summary of the State of the Art*. Kluwer Academic Publishers, 1994. 19

[86] J. B. Dennis and G. R. Gao. **Multithreaded architectures: Principles, projects, and numbers**. Technical report, School of Computer Science, McGill University, Montreal, Quebec, CA, 1994. 19

[87] C. Kim and J. L. Gaudiot. *Dataflow and Multithreaded Architectures*. Wiley, New York, 1997. 19

[88] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. **Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture**. In *International Symposium on Computer Architecture (ISCA)*, pages 422–433, 2003. 19, 20

[89] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. Mcdonald, and C. R. Moore. **TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP**. *ACM Transaction on Architecture and Code Optimazation (TACO)*, **1**(1):62–93, 2004. 19, 20

[90] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. **WaveScalar**. In *International Symposium on Microarchitecture (MICRO)*, pages 291–302, 2003. 19, 21

[91] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. Eggers. **The WaveScalar architecture**. *ACM Transaction on Computer Systems (TOCS)*, **25**(2):4:1–4:54, 2007. 19

[92] K. M. Kavi, R. Giorgi, and J. Arul. **Scheduled dataflow: Execution paradigm, architecture, and performance evaluation**. *IEEE Transactions on Computers*, **50**:834–846, 2001. 19, 21

[93] P. Evripidou and C. Kyriacou. **Data driven network Of workstations (D^2NOW)**. *Journal of Universal Computer Science (JUCS)*, **6**(10):1015–1033, 2000. 19

[94] P. Evripidou. **D3-Machine: A decoupled data-driven multithreaded architecture with variable resolution support**. *Parallel Computing*, **27**:1197–1225, 2001. 19

[95] C. Kyriacou, P. Evripidou, and P. Trancoso. **Data-driven multithreading using conventional microprocessors**. *IEEE Transaction on Parallel and Distributed Systems (TPDS)*, **17**(10):1176–1188, 2006. 19, 21

[96] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu. **Tartan: Evaluating spatial computation for whole program execution**. In *International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 163–174, 2006. 19, 20

[97] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. **Conservation cores: Reducing the energy of mature computations**. In *International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–218, 2010. 19, 20

[98] V. Govindaraju, C. Ho, and K. Sankaralingam. **Dynamically specialized datapaths for energy efficient computing**. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2011. 19, 20

[99] V. Govindaraju, C. H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. **DySER: Unifying functionality and parallelism specialization for energy-efficient computing**. *IEEE Micro*, **32**:38–51, 2012. 19

[100] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M. Valero. **Kilo-instruction processors: Overcoming the memory wall**. *IEEE Micro*, **25**(3):48–57, 2005. 20

[101] R. S. Nikhil, G. M. Papadopoulos, and Arvind. ***T: A multithreaded massively parallel architecture**. In *International Symposium on Computer Architecture (ISCA)*, pages 156–167, 1992. 21

[102] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. Eicken. **TAM: A compiler controlled threaded abstract machine**. *Journal of Parallel and Distributed Computing (JPDC)*, **18**(3):347–370, 1993. 21

[103] J. Strohschneider, B. Klauer, S. Zickenheimer, and K. Waldschmidt. **ADARK: A fine grain dataflow architecture with associative communication network**. In *EUROMICRO Conference*, pages 445–450, 1994. 21

[104] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryky, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnany, A. Marquez, S. Merali, S. S. Nemawarkarz, P. Panangaden, X. Xue, and Y.Zhu. **A design study of the EARTH multiprocessor**. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 59–68, 1995. 21

[105] R. S. Nikhil. **Can dataflow subsume von Neumann computing?** In *International Symposium on Computer Architecture (ISCA)*, pages 262–272, 1989. 21

[106] E. A. Lee and D. G. Messerschmitt. **Synchronous data flow**. *Journal of IEEE*, **75**(9):1235–1245, 1987. 21

[107] D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C. Q. Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U. M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, and J. Andrews. **The Cedar system and an initial performance study**. In *International Symposium on Computer Architecture (ISCA)*, pages 213–223, 1993. 21

[108] V. Vee and H. Wen-Jing. **A scalable and efficient storage allocator on shared-memry multiprocessors**. In *IEEE International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, 1999. 26

[109] **OpenMP Application Program Interface, Version 3.1, Retrieved 2014-02-06**. In *openmp.org*, 2011. 26

[110] K. Wooyoung and M. Voss. **Multicore desktop programming with Intel threading building blocks**. *IEEE Software*, **28**(1), 2011. 26

[111] **NVIDIA: NVIDIA CUDA Compute Unified Device Architecture-Programming Guide, v. 2. 0, www. nvidia. com/object/cuda-home. html, 2009.** 26

[112] J. E. Stone, D. Gohara, and G. Shi. **OpenCL: A parallel programming standard for heterogeneous computing systems**. *Computing in Science and Engineering*, **12**(3), 2010. 26

[113] M. C. Rinard, D. J. Scales, and M. S. Lam. **Jade: A high-level, machine-independent language for parallel programming**. *Computer*, **26**(6):28–38, 1993. 26, 34

[114] 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 (ScalA)*, pages 19–20, 2011. 26, 34

[115] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. **Sequoia: Programming the memory hierarchy**. In *Supercomputing (SC)*, 2006. 26

[116] M. D. McCool and B. D'Amora. **Programming using RapidMind on the Cell BE**. In *ACM/IEEE Conference on Supercomputing (SC)*, 2006. 26

[117] M. D. McCool, K. Wadleigh, B. Henderson, and H. Y. Lin. **Performance evaluation of GPUs using the RapidMind development platform**. In *ACM/IEEE Conference on Supercomputing (SC)*, 2006. 26

[118] I. Christadler and V. Weinberg. **Facing the multicore-challenge**. chapter RapidMind: Portability across architectures and its limitations, pages 4–15. 2010. 26

[119] J. C. Jenista, Y. H. Eom, and B. Demsky. **OoOJava: An out-of-order approach to parallel programming**. In *USENIX Conference on Hot Topic in Parallelism (HotPar)*, pages 11–11, 2010. 26, 34

[120] **OpenMP Application Program Interface, Version 4.0, Retrieved 2014-02-06**. In *openmp.org*, 2013. 26, 34

[121] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta. **ClusterSs: A task-based programming model for clusters**. In *International Symposium on High Performance Distributed Computing (HPDC)*, pages 267–268, 2011. 26

[122] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta. **A high-productivity task-based programming model for clusters**. *Concurrency and Computation: Practice and Experience*, **24**(18):2421–2448, 2012. 26

[123] 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*, **21**(2):173–193, 2011. 26, 27, 34, 35

[124] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta. **Productive cluster programming with OmpSs**. In *International Conference on Parallel Processing (Euro-Par)*, pages 555–566, 2011. 26, 27, 34, 83

[125] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta. **Hierarchical task-based programming with StarSs**. *J. High Performance Computing Application*, **23**(3). 27

[126] V. Marjanovic, J. Labarta, E. Ayguade, and M. Valero. **Overlapping communication and computation by using a hybrid MPI/SMPSs approach**. In *ACM International Conference on Supercomputing (ICS)*, pages 5–16, 2010. 27

[127] G. Tzenakis, K. Kapelonis, M. Alvanos, K. Koukos, D. S. Nikolopoulos, and A. Bilas. **Tagged procedure calls (TPC): Efficient runtime support for task-based parallelism on the cell processor**. In *HiPEAC*, Lecture Notes in Computer Science, pages 307–321, 2010. 27

[128] M. Gonzalez, J. Balart, A. Duran, X. Martorell, and E. Ayguade. **Nanos Mercurium: A research compiler for OpenMP**. In *European Workshop on OpenMP*, 2004. 27, 83

[129] G. Al-Kadi and A. S. Terechko. **A hardware task scheduler for embedded video processing**. In *International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, pages 140–152, 2009. 34, 35

[130] M. S. Lam and M. C. Rinard. **Coarse-grain parallel programming in Jade**. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 94–105, 1991. 34

[131] M. C. Rinard, D. J. Scales, and M. S. Lam. **Heterogeneous parallel programming in Jade**. In *Conference on Supercomputing*, pages 245–256, 1992. 34

[132] S. Kumar, C. J. Hughes, and A. Nguyen. **Carbon: Architectural support for fine-grained parallelism on chip multiprocessors**. In *International Symposium on Computer Architecture (ISCA)*, pages 162–173, 2007. 34

[133] J. Hoogerbrugge and A. Terechko. **A multithreaded multicore system for embedded media processing**. *Transaction on High-performance Embedded Architectures and Compilers (THEA)*, **3**(2), 2011. 34

[134] M. Sjalander, A. Terechko, and M. Duranton. **A look-ahead task management unit for embedded multi-core architectures**. In *Conference on Digital System Design (DSD)*, pages 149–157, 2008. 34, 35

[135] J. Castrillon, D. Zhang, T. Kempf, B. Vanthournout, R. Leupers, and G. Ascheid. **Task management in MPSoCs: An ASIP approach**. In *International Conference on Computer-Aided Design (ICCAD)*, pages 587–594, 2009. 34

[136] A. C. Nacul, F. Regazzoni, and M. Lajolo. **Hardware scheduling support in SMP architectures**. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 642–647, 2007. 34

[137] S. Park. **A hardware operating system kernel for multi processors**. *IEICE Electronics Express*, **5**(9):296–302, 2008. 34

[138] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. **NVIDIA Tesla: A unified graphics and computing architecture**. *IEEE Micro*, **28**(2):39–55, 2008. 34, 35

[139] Saez and Vila. **A hardware scheduler for complex real time system**. In *IEEE International Symposium Industrial Electronics (ISIE)*, 1999. 35

[140] R. Kalra and R. Lysecky. **Configuration locking and schedulability estimation for reduced reconfiguration overheads of reconfigurable systems**. *IEEE Transaction Very Large Scale Integrated System*, **18**(4):671–674, 2010. 35

[141] J. Noguera and R. M. Badia. **System-level power-performance trade-offs in task scheduling for dynamically reconfigurable architectures**. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 73–83, 2003. 35

[142] J. Noguera and R. M. Badia. **Multitasking on reconfigurable architectures: Microarchitecture support and dynamic scheduling**. *ACM Transaction Embedded Computing System*, **3**(2):385–406, 2004. 35

[143] C. Meenderinck and B. Juurlink. **A case for hardware task management support for the StarSs programming model**. In *Conference on Digital System Design (DSD)*, pages 347–354, 2010. 35

[144] C. Meenderinck and B. Juurlink. **Nexus: Hardware support for task-based programming**. In *Conference on Digital System Design (DSD)*, pages 442–445, 2011. 35

[145] Fahimeh Yazdanpanah, D. Jimenez-Gonzalez, C. Alvarez-Martinez, Y. Etsion, and R. M. Badia. **FPGA-based prototype of the Task Superscalar architecture**. In *7th HiPEAC Workshop of Reconfigurable Computing (WRC)*, 2013. 52

[146] **BSC Application Repository, BAR, url: https://pm.bsc.es/projects/bar. Retrieved 2014-02-06**. In *Barcelona Supercomputing Center (BSC)*, 2014. 83

[147] Peter K. Pearson. **Fast Hashing of Variable-length Text Strings**. *Commun. ACM*, **33**(6):677–680, 1990. 92