Simultaneous multithreading: Operating system perspective

Vyacheslav Rubinfine

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Simultaneous Multithreading:

Operating System Perspective

By

Vyacheslav Rubinfine

A Thesis Submitted

in

Partial Fulfillment of the
Requirements for the Degree of
MASTERS OF SCIENCE

in

Computer Engineering

Approved by:

Committee Principle: _______________________________ Date: 11-5-2002
Dr. Muhammad Shaaban

Committee Member: _______________________________ Date: Nov 5, 2002
Dr. Roy S. Czernikowski

Committee Member: _______________________________ Date: Nov 5, 2002
Dr. Hans-Peter Bischof

Department of Computer Engineering
College of Engineering
Rochester Institute of Technology
Rochester New York, USA
Simultaneous Multithreading:
Operating System Perspective

I, Vyacheslav Rubinfine, hereby grant permission to any individual or organization to reproduce this thesis in whole or in part for non-commercial and non-profit purposes only.

Vyacheslav Rubinfine
September 27, 2002
Abstract

Developing CPU architecture is a very complicated, iterative process that requires significant time and money investments. The motivation for this work is to find ways to decreases the amount of time and money needed for the development of hardware architectures.

The main problem is that it is very difficult to determine the performance of the architecture, since it is impossible to take any performance measurements until upon completion of the development process. Consecutively, it is impossible to improve the performance of the product or to predict the influence of different parts of the architecture on the architecture's overall performance. Another problem is that this type of development does not allow for the developed system to be reconfigured or altered without complete re-development.

The solution to the problems mentioned above is the software simulators that allow researching the architecture before even starting to cut the silicon. Simultaneous multithreading (SMT) is a modern approach to CPU design. This technique increases the system throughput by decreasing both total instruction delay and stall times of the CPU. The gain in performance of a typical SMT processor is achieved by allowing the instructions from several threads to be fetched by an operating system into the CPU simultaneously. In order to function successfully the CPU needs software support.

In modern computer systems the influence of an operating system on overall system performance can no longer be ignored. It is important to understand that the union of the CPU and the supporting operating system and their interdependency determines the overall performance of any computer system. In the system that has been implemented on hardware level such analysis is impossible, since the hardware system is neither flexible nor configurable. However, in the SMT architecture, the system is capable of performing some useful work even if a task has generated an error. A wide range of simulators is described in the literature, and a lot of them are publicly accessible.

The main goal of this work is to modify an existing SIMOS/Topsy simulator to achieve a simple, configurable, publicly accessible SMT SIMOS/Topsy simulator that must also include an SMT Topsy. The simulator should demonstrate the fetching
process of the SMT MIPS, as well as scheduling aspects of the CPU and the operating system integrated environment.

This work covers a broad range of aspects, among which are:

1) Completion of SMT MIPS and SMT Topsy specifications;
2) Integration of MXS into SIMOS/Topsy;
3) Modifications to the fetching unit of MXS that allow to support SMT;
4) Addition of SMT support to Topsy;

This work uses Topsy/R4000 simulator developed at Swiss Federal Institute of Technology, and the MXS (R10000) part of the SimOS simulator developed at Stanford University. Development process utilizes C high-level language, Intel and MIPS assembly languages.

The result of this work is a development of a complete computer system software simulator. The simulator allows taking performance measurements and reconfiguration of SMT Topsy and the fetching unit of the SMT MXS. The simulator is modular: that is any of its parts can be substituted with other parts that perform similar functionality. It also means that the whole simulator can be integrated into a larger scale simulation project.

The development of this simulator significantly decreases the amount of time and money needed for the development of hardware architectures and provides new ways in researching the influence of an operating system on the performance of the computer system as a whole.
Acknowledgements

- This work is dedicated to my family that has supported and inspired me during the difficult time I have spent at school.

- I am grateful to my wife Elena for spending countless hours doing proofreading and other paperwork. Thanks go to my friend Slava Rozman for his help in completion of successful installation and compilation of the software.

- I am grateful to Professor Muhammad E. Shaaban at the Computer Engineering department of the Rochester Institute of Technology for raising my interest in to the topic and patiently answering all my questions related to various computer architectures and operating systems. I wish to thank him for giving me the opportunity to work on this project and shaping my learning with his valuable hardware and software insight.
Trademarks

This work is based on the following software:

- SimOS Trademark of Stanford University
- Topsy Trademark of UIT
# Table of Contents

List of Figures ........................................................................................................... 3  
List of Tables ............................................................................................................. 5  
Glossary of Terms ...................................................................................................... 6  
Chapter 1. Introduction .............................................................................................. 8  
  1.1. Basic architectures and techniques ................................................................. 8  
  1.2. Pipeline and hazards ....................................................................................... 10  
  1.3. Superscalar architecture and SMT ................................................................. 15  
  1.4. Overview of subsequent chapters .................................................................... 16  
Chapter 2. Developing SMT SimOS/Topsy environment ........................................... 17  
  2.1. Basic SimOS/Topsy environment .................................................................... 17  
  2.2. Switching to SMT SimOS/Topsy environment .............................................. 18  
    2.2.1. Problems associated with basic SimOS/Topsy environment ............... 19  
    2.2.2. SMT SimOS/Topsy environment ............................................................ 19  
    2.2.3. Work cycle of the SMT SimOS/Topsy environment ......................... 21  
  2.3. Annotations vs. Performance Counters ......................................................... 23  
Chapter 3. Developing SMT MXS CPU model ......................................................... 26  
  3.1. General Description ....................................................................................... 27  
  3.2. Instruction Set ............................................................................................... 29  
  3.3. Pipeline ......................................................................................................... 34  
  3.4. Caches .......................................................................................................... 38  
    3.4.1. Primary Instruction Cache ..................................................................... 38  
    3.4.2. Primary Data Cache .............................................................................. 38  
    3.4.3. Secondary Cache .................................................................................. 39  
    3.4.4. Cache Algorithms .................................................................................. 39  
    3.4.5. Virtual Address Translation and Pages .............................................. 40  
  3.5. Fetching Unit of SMT MXS CPU model ....................................................... 41  
    3.5.1 Basic MXS Fetching Unit ....................................................................... 41  
    3.5.2 SMT MXS Fetching Unit ....................................................................... 42  
    3.5.3 Enhanced SMT MXS Fetching Unit ....................................................... 43  
  3.6. Register Renaming Unit and Special Techniques ......................................... 47  
    3.6.1. Operating System Support for Register Renaming ......................... 49  
    3.6.2. Compiler Support for Register Renaming ........................................... 51  
    3.6.3. SMT MXS Support for Register Renaming ........................................ 53  
  3.7. SMT Issue Queues ......................................................................................... 55  
    3.7.1 Integer Queue ......................................................................................... 55  
    3.7.2 Floating-point Queue ............................................................................ 56  
    3.7.3. Address Queue ..................................................................................... 56  
  3.8. Branch Prediction Policies ............................................................................ 57  
  3.9. Branch Prediction Unit of SMT SimOS ....................................................... 65  
  3.10. WRITE Branch Prediction Policy ............................................................... 72  
  3.11. Exception Handling in SMT MXS .............................................................. 72  
    3.11.1. Cold Reset Exception ....................................................................... 74  
    3.11.2. Soft Reset Exception ........................................................................ 75  
    3.11.3. Address Error Exception .................................................................. 75
List of Figures

Figure 1. Basic SimOS/Topsy environment. Page 17.

Figure 2. SMT SimOS/Topsy environment. Page 19.

Figure 3. Work cycle of SMT SimOS/Topsy environment. Page 21.

Figure 4. Viewing data with RIT Viewer application. Page 24.

Figure 5. Port table entry. Courtesy of Dominik Madoń, Eduardo Sánchez, Stefan Monnier, Computer Science Department, Yale University. Page 32.

Figure 6. Modular structure of Topsy operating system. Courtesy of George Fankhauser, Christian Conrad, Eckart Zitzler, Bernhard Pattner, Computer Engineering and Networks Laboratory, ETH Zürich. Page 89.

Figure 7. Bootstrapping process of SMT Topsy operating system. Courtesy of George Fankhauser, Christian Conrad, Eckart Zitzler, Bernhard Pattner, Computer Engineering and Networks Laboratory, ETH Zürich. Page 90.

Figure 8. Kernel thread messages queue organization of Topsy operating system. Courtesy of George Fankhauser, Christian Conrad, Eckart Zitzler, Bernhard Pattner, Computer Engineering and Networks Laboratory, ETH Zürich. Page 97.

Figure 9. Kernel thread messages queue organization of SMT Topsy operating system. Page 98.

Figure 10. SMT MXS fetching process. Page 106.

Figure 11. Kernel-based SMT MXS fetching process. Page 107.
Figure 12. BasicMXS Fetching Unit performance. Page 114.

Figure 13. SMT MXS Fetching Unit performance: 2 contexts. Page 115.

Figure 14. SMT MXS Fetching Unit performance: 4 contexts. Page 116.

Figure 15. SMT MXS Fetching Unit performance: 8 contexts. Page 117.

Figure 16. Comparing none-SMT and SMT MXS Fetching Units. Page 117.

Figure 17. Comparing none-SMT and SMT MXS Fetching Units. Page 118.

Figure 18. Basic Topsy scheduler. Page 118.

Figure 19. SMT Topsy scheduler. Page 119.

Figure 20. Basic Topsy scheduling process. Page 120.

Figure 21. SMT Topsy scheduling process. Page 121.

Figure 22. SMT Topsy scheduling process. Page 122.

Figure 23. Comparing SMT Topsy scheduling processes, uniform and none-uniform tasks. Page 123.
List of Tables

Table 1. *Main configuration parameters of SMT SimOS/Topsy environment.* Page 20.

Table 2. *Viewing data with RIT Viewer application.* Page 24.


Table 4. *Instructions compatibility in SMT microprocessor.* Page 37.

Table 5. *Operational modes of SMT MIPS microprocessor.* Page 40.

Table 6. *SMT Coprocessor 0 instructions.* Page 79.

Table 7. *Number and types of registers in SMT MXS CPU model.* Page 127.

Table 8. *Parameters related to register renaming and load/store operations in SMT MXS simulator.* Page 128.


Table 10. *Branch prediction parameters of SMT MXS simulator.* Page 129.

Table 11. *Latencies of the various operations of the functional units of SMT MXS simulator.* Page 129.


Glossary of Terms

- **Superscalar** - CPU architecture that can fetch, execute and complete more that one instruction in parallel.
- **ANDES** - Architecture with Non-sequential Dynamic Execution Scheduling
- **Dynamic instruction scheduling** – CPU issues an instruction as soon as all its operands become available and the required execution unit is not busy.
- **Out-of-order execution** – in Superscalar CPU instructions can be executed not in program order..
- **Speculative branching** – the CPU executes the branch based on the previous history of the branch or based on its own fetching policy, since there is not enough time for the CPU to calculate the real branch address due to the CPU fetch bandwidth.
- **Non-blocking cache** – type of cache, in which cache misses do not stall the CPU.
- **Wrong-path instructions** - instructions that were fetched as the result of branch misprediction.
- **Pipeline** – the process of dividing of each instruction into a sequence of simpler suboperations (stages), each of which is performed by a separate hardware unit.
- **Pipeline latency** - the number of cycles between the time an instruction is issued and the time the dependent instruction (the instruction which uses its result as an operand) can be issued.
- **Precise exception handling** - technique that allows handling of an exception at the moment it occurs. A typical processor design implements precise exceptions by: 1) identifying the instruction, which caused the exception; 2) preventing the exception-causing instruction from graduating; 3) aborting all subsequent instructions.
- **Strong ordering** - in-order instruction graduation maintained, though the instructions may be issued and executed out-of-order.
- **Register renaming** - logical registers specified in the operands are mapped to physical registers (integer and floating point registers are mapped separately).
- **Forwarding logic** - hardware that makes the result of an instruction available for subsequent instructions before the instruction graduates.
- **Dead-register distance** – the number of cycles passed between a register’s last use and its redefinition.
- **HAL** – hardware abstraction layer.
- **SMT** – simultaneous multithreading. The SMT-capable CPU can process instructions from several processes simultaneously. The SMT-capable operating system can have several processes in running state simultaneously.
- **SIMOS** - complete machine simulator environment developed at Stanford University.
- **SMT SIMOS** – complete machine simulator environment that supports SMT.
- **Topsy** – simple Teachable Operating System developed at Swiss Federal Institute of Technology.
- **SMT Topsy** – Topsy that supports SMT.
- **MIPS** – CPU type. Can be referred to in conjunction with R3000, R4000, R10000, or R12000 processors.
- **SMT MIPS** – MIPS processor model that supports SMT.
- **SIMOS/Topsy** – integrated environment that includes both SIMOS and TOPSY.
- **SMT SIMOS/Topsy** – SIMOS/Topsy that supports SMT.
- **MXS** – part of SIMOS that implements R10000 CPU.
- **SMT MXS** – MXS that supports SMT.
Chapter 1. Introduction

- Section 1.1 illustrates basic architectures and techniques that are used in CPU design.
- Section 1.2 describes the issues related to pipelining and problems associated with this technique.
- Section 1.3 represents superscalar and SMT CPU architectures.
- Section 1.4 provides an overview of the subsequent chapters.

1.1. Basic architectures and techniques

During the second half of last century computer technology has made incredible progress. The microprocessor performance growth improved at the rate of roughly 35% per year [1]. Any modern computer architecture has to satisfy various requirements, such as:

1) Instruction set architecture should refer to the actual visible user set of assembly instructions. At the same token, it has to be minimal, yet flexible enough, to allow the user to manipulate all aspects of the contemporary CPU.
2) Modern memory hierarchy should not only include physical implementation of all the levels of caches, but also effective memory management strategies. These strategies imply, but not limited to increasing the hit/miss cache ratio, memory consistency and access and managing persistent media.
3) Performance should include such aspects as instructions scheduling and execution; hardware interrupts.
4) Operating systems specifics should include processes scheduling, process memory management, software interrupts, file systems and other aspects.

In light of the complicated tasks mentioned above, and, in particular, of the CPU performance requirements, this work addresses the following issues:

1) General definition of the main principals of high performance CPUs.
2) Formulation of the hardware and software requirements for the CPU model along with the principals of high performance CPU.
3) Determination of the factors that allow most efficient interaction between a UNIX-type operating system and the high performance CPU.

4) Identification and implementation of a set of changes to the operating system that allows reliable use of SMT features.

5) Foundation for future research and development in the area of computer engineering consistent with the findings presented in this work.

The formula presented below is used to calculate performance of any CPU:

$$P = IC \times CPI \times T;$$

Where: IC - instruction counter, indicates the total number of instructions in a program;
CPI - cycles per instruction, indicates the number of machine (command) cycles per instruction;
T - clock cycle time that is individually measured for each type of CPU.

This formula determines that the CPU execution time is equally dependent on all of the three parameters: IC, CPI and T.

Since T is a constant hardware characteristic of each particular type of CPU and does not change depending on process parameters and organization, this work is mostly dedicated to the other two component of the equation: IC and CPI.

The stack-based CPU designs suggested a good IC * CPI ratio by allowing all operations to utilize the stack. In that case, the CPU was waiting on the first operand to be pushed onto the stack, then for the second operand to be entered likewise, the operation to occupy the top of the stack, and, at last, for execution to take place. The accumulator-based architectures employed slightly different approach. One operand was loaded into a special purpose register, also called accumulator, and the other operand was accessed directly from memory or loaded into another temporary location and accessed from there. Those two instruction set architectures were named register-register and register-memory architectures accordingly. They both presented advantages, as well as disadvantages:

1) The advantage of register-register architecture is that it unifies the CPI by presenting simple fixed-length instructions, as well as an efficient code generation model. The
disadvantage of the register-register architecture is that instructions cannot directly refer to memory; the IC for this architecture is higher than for the register-memory architecture.

2) The advantage of the register-memory architecture presents good code density, which yields the good IC. Because the instructions contain direct memory references, data can be accessed without loading first. One of the disadvantages of the register-memory architecture is that even though the CPI parameter for this architecture is lower than for the register-register architecture, the operands are not equal in size. Another disadvantage of the register-memory architecture is that encoding a register number and a memory address in each instruction may restrict the number of registers.

The listed above architectures presented a significant freedom to computer engineers and designers. However, neither of the architectures could produce a significant boost in the CPU performance. The reason for that was that the CPU executed all the instructions (no matter from where they were taken) sequentially. Therefore, the total execution time depended linearly on the number of the instructions introduced by a particular process. On the other hand, analysis of processes showed that a typical process had a large number of independent instructions. The CPU designers faced a great challenge in developing technique that allows multiple instructions to be simultaneously issued to the CPU.

1.2. Pipeline and hazards

The basic idea of pipelining (the new and promising way of issuing multiple instructions to the CPU) is simple: similar to an assembly line, an instruction execution is broken up into several steps. Different stages complete particular parts of various instructions in parallel. Each of these steps is named either a pipe stage or a pipe segment. This instruction execution time can be calculated as a product of the number of pipeline stages (the pipeline depth) and the length of the slowest pipeline stage, assuming that there are no problems during the execution that may postpone the execution of the instruction. Such time is required to measure the ideal performance of the pipeline. In real life, however, the execution time of an instruction in the pipeline is significantly different from its ideal counterpart: an instruction making its way through the pipeline can be stalled or even stopped.
due to a number of factors. These factors are called hazards and categorized into several groups, depending on their nature. The following paragraphs provide more details on the pipeline hazards.

Any hazard prevents the next instruction in the instruction stream from executing during its designated clock cycle. There are three classes of hazards [1]: 1) structural hazards that arose from resource conflicts when the hardware could not provide so called “support orthogonality”. In other words structural hazards occurred when not all possible combinations of instruction could be supported by the hardware. 2) data hazards that arose when the instruction depends on the results of the previous instruction in a way that was exposed by overlapping of instructions in the pipeline. 3) control hazards caused by branches and other instructions that changed the PC. The following is a more detailed discussion on the structural hazards.

The most common instances of structural hazards occurred when some functional units were not fully pipelined. In technical terms it meant that a particular functional unit or the number of functional units could not execute the instructions at the rate of one per clock cycle. According to [1], when the sequence of instructions encountered this hazard, the pipeline stalled one of the instructions until the required unit became available. Some pipelines contained mixed instruction and data streams, which increased the average number of pipeline stalls. As the result, the majority of modern machines have separate instruction and data pipelines. Another solution in preventing structural hazards is the duplication of hardware units. For more details on structural hazards please refer to [1]. The next and more serious type of hazards is the data hazard. There are many situations when a particular instruction got a wrong operand as the result of non-finished execution of its predecessor, or an operation not being overwritten before the instruction waiting for this result had a chance to read it. Data hazards can be described as one of the three types, depending on the order, in which read and write accesses occurs in the instructions. For two instructions A and B, with A occurring before B, the possible data hazards are: 1) RAW (read after write) – B attempted to read a source before A wrote it, 2) WAW (write after write) – B tried to write an operand before it was written by A, and 3) WAR (write after read) – B proceeded with writing a destination before it was read by A. Several techniques were developed to reduce the effect of data hazards, including forwarding, compiler scheduling etc. Additional special
techniques will be mentioned further down as a part of presenting instruction-level parallelism in pipelining. As it is shown in [1], the data hazards caused a great deal of performance loss in the pipeline. However, control hazards are always considered to be the main source of the loss of the CPU performance.

The simplest way to deal with the control hazards is to stall the pipeline as soon as the hazard was detected. In other words, the pipeline had to be stalled until the new value of PC was calculated. However, in order to really solve the problem, the CPU designers had to find answers to the following questions: 1) how to determine whether the branch was taken or not taken earlier in the pipeline and 2) how to compute the taken PC (i.e., the address of the branch target) earlier. The length of the control hazard was called a branch delay. In general, the deeper pipeline, the worse the branch penalty was in clock cycles. Here several choices of reducing pipeline branch delays will be considered.

As it was presented above, the simplest scheme to handle branches is to stall the pipeline. A higher performance, but somewhat more complex, scheme was to predict the branch as not taken, simply allowing the pipeline to proceed as if the branch were never executed. An alternative scheme is to predict the branch as taken and always continue the execution at the branch target instruction address. Yet another scheme was developed that was called delayed branch. This scheme placed the sequential successors of the branch in the branch-delay slots. These instructions were executed whether or not the branch was taken. The modern and the most practical use of this scheme is that branches have a single instruction delay slot. There were many other techniques the computer engineers introduced to prevent control hazards. For more details on prevention of control hazards please refer to [1].

The listed above explained how pipelining could overlap the execution of instructions when the instruction were independent of one another. Several techniques were introduced to extend the pipelining ideas by increasing the amount of parallelism introduced by many instructions.

Consider the case with three instructions A, B, C where the instruction C did not depend on either A or B, than the execution of C was postponed until it commits the results to memory. To remove this constraint, the concept of out-of-order execution was introduced. This concept can be briefly described as follows: after an instruction had been fetched, in the
presence of structural and data hazards, the instruction had to wait for its operands to become ready. Only after both operands had been present was the instruction ready to move to the execution stage. It clearly showed that in some cases an instruction had to occupy CPU units for significant time; the whole pipeline had to be stalled while some instructions were actually able to execute. The logical solution to this problem was to split the decode stage into two: issue and read operands stages. If there were no structural hazards, the instruction would be checked against the readiness of its operands, and if any operand were not ready due to the data hazard, the instruction would not be issued to an appropriate CPU unit, but into the queue. Moreover, since the execution phase might span a few CPU cycles, the CPU designers distinguished between begin and complete execution phases. The technique of introducing multiple instructions to CPU execution units was named “dynamic scheduling” and became extremely popular in a very short time.

The following subsections briefly describe different techniques for dynamic scheduling, their advantages and disadvantages, as well as inability to solve certain problems that arose for the computer engineers at that time.

**Scoreboarding** [1] was used in those cases when the CPU executes instructions in an out-of-order manner in the presence of sufficient hardware resources and absence of data hazards. As described in [1], every instruction went through the scoreboard, where a record of all data dependencies was constructed. This action implied WAW hazard checking, as well as the availability of the appropriate functional units. Only if no other instruction wanted to write the result to the output register, and the unit was free, was the instruction issued. Then the scoreboard determined when the instruction could read its operands and begin execution. The instruction was executed, and the scoreboard was notified upon its completion. If no WAR hazards were detected, the scoreboard then wrote the result in memory [1]. The disadvantages of scoreboarding were: 1) scoreboarding did not allow forwarding, since it could only apply operands after both of them had been placed into its registers; 2) scoreboarding was getting all the information from communication with functional units, which was limited by the number of available operands/result buses in a register file.

Perhaps, the most significant limitations of scoreboarding, according to [1], were the amount of parallelism within an executed process (the number of independent instructions
that could be found), and the number of scoreboard entries (scoreboard window, i.e. the number of instructions that could be actually placed for execution at any time). Because dynamically scheduled pipeline introduced more data dependencies, another technique, known as register renaming, or the Tomasulo Approach, was introduced.

**Tomasulo Approach** was used primarily in those architectures; where the number of functional units was not significant, though the units themselves were pipelined. Instead of compiler register renaming, the WAW / WAR dependencies elimination was achieved by use of reservation stations. The reservation stations buffered the operands of instructions that were waiting to be issued by the issuing logic [1]. First, the reservation station saved the operand eliminating the need to get it from the register. Second, any pending instructions could use reservations stations as their input. Finally, the WAW hazards could not happen due to the fact that only the last write operation to the reservation station actually caused the register update.

As specified by [1], there were two significant advantages in the organization of Tomasulo's schema: 1) hazards detection and execution control were distributed, since the reservation station at every functional unit decided when an appropriate instruction could execute at the unit; 2) the results were passed from the reservation stations directly, eliminating the need of use of registers and associated data hazards. As the result of all the aspects mentioned above, the Tomasulo’s technique was useful on single-issue pipelines with limited number of registers.

Even though a number of other techniques were also developed to improve CPU performance, none of them could increase the number of executed instructions per cycle to any larger value than 1. This problem remained unresolved for a long time, and was successfully overcome by the introduction of the wide population of Superscalar CPUs, VLIW CPUs, super pipelined and vector CPUs, as well as multiprocessor designs.
1.3. Superscalar architecture and SMT

Consistent with the definition of a base scalar processor [2], simple operation and instruction latencies were all equal to one, the Superscalar processor issued multiple instructions and generated multiple results per cycle. A vector processor executed vector instructions on the arrays of data such that each instruction invoked a string of parallel operations. This was ideal for pipelined architecture with one result per cycle. In theory, according to [3], a Superscalar processor may reach the same performance as a machine with vector hardware. If a Superscalar machine could issue a fixed-point, floating-point, load and branch, all in one cycle, the effect would be the same as from the use of vector load chained with vector add machine. A typical VLIW (very long instruction word) [2] machine had instruction words hundreds of bits in length. Like in a Superscalar processor, multiple functional units were used concurrently. All units shared the common register file.

The concept of micro coding, as stated by [2], implied the use of different fields of the long instruction word to carry the operation codes in the order to be dispatched to different functional units. Indeed, the code written in conventional short instructions had to be combined to use in VLIW computers. However, the vector machines are beyond the scope of this work, and for this reason, this work will focus on Superscalar architecture only.

The issue utilization in the superscalar architecture, i.e. the percentage of issue slots that were filled each cycle, along with the cause for every empty slot was measured using the SPEC benchmark. If the next instruction could not be assigned during the same cycle as the current instruction, then the remaining issue slots at this cycle, as well as issue slots of idle cycles during the execution of the current instruction, were assigned to the causes of the delay. In case of overlapped causes, the longest delay was assigned. The results showed that the functional units were highly underutilized. On average, according to [2], the architecture reached only 1.5 instructions per cycle, with a full 8-way issuing schema. Also, it was clear that there was no any dominant source of wasted issue bandwidth. Although there were dominant items in individual benchmarked applications, the dominant cause was different in each case. All above listed major techniques of reducing the waste were applied. None of them, though, produced a satisfactory increase in performance, since they only attacked specific types of latencies. Therefore, as was shown by [2], even the 8-way Superscalar CPU
was not capable of complete elimination of either vertical waste (completely idle cycles) or horizontal waste (unused issue slots in a non-idle cycle).

### 1.4. Overview of subsequent chapters

- Chapter 2, *Developing SimOS/Topsy Environment*, provides basic information about the initial integrated model for the development. The chapter describes necessary modifications to convert the initial integrated environment to the SMT MXS integrated environment.
- Chapter 3, *Developing SMT CPU Model* introduces theoretical foundation and a set of requirements for SMT CPU. The chapter includes several implementation issues, along with the results of instruction fetching by the SMT MXS fetching unit.
- Chapter 4, *Developing SMT Topsy* describes a set of changes to the modern operating system (Topsy) to accommodate the SMT scheduling. This chapter also provides particular implementation details of the SMT MXS scheduler.
- Chapter 5, *SMT MXS SimOS/Topsy Installation*, describes the process of installation of the SMT MXS SimOS/Topsy integrated environment on a user’s PC (Linux platform).
- Chapter 6, *SMT MXS SimOS/Topsy Configuration*, describes the process of configuration of the SMT MXS SimOS/Topsy integrated environment, after it was installed on a user’s PC (Linux platform).
Chapter 2. Developing SMT SimOS/Topsy environment

- Section 2.1 discusses the basic SimOS/Topsy integrated environment and its work cycle.
- Section 2.2 introduces a few problems associated with the basic environment.
- Section 2.3 illustrates advantages of the SMT SimOS/Topsy and describes its working cycle.
- Section 2.4 demonstrates a new Performance Counters concept that is used to monitor the work cycle of the SMT SimOS/Topsy integrated environment.

2.1. Basic SimOS/Topsy environment

This section provides an overview of the basic integrated environment, which structure is shown in Figure 1.

![Figure 1. Basic SimOS/Topsy environment](image)

In order to be executed by the SimOS simulator, Topsy has to be compiled and linked using MIPS cross-compiler. The output file, topsy.ecoff, is then loaded into the SimOS simulator. The utilization of the SimOS environment in the basic implementation is limited to the use of R4000 CPU model. The parts of the simulator that implement primary and
secondary caches, as well as the implementation of the virtual machines and CPU vectorization, are disabled.

2.2. Switching to SMT SimOS/Topsy environment

As indicated in [3], SMT is a latency-tolerant CPU architecture that is able to execute multiple instructions from multiple threads during each cycle. This ability to issue instructions from different threads provides for a better utilization of execution resources by converting thread-level parallelism into instruction-level parallelism. A number of previous studies have established that SMT is an effective instrument for increasing performance on a variety of workloads, and, at the same time, it provides good performance for single-threaded applications.

As it was shown in [1], at the hardware level, SMT usually serves as an extension to any out-of-order superscalar. A good example of this would be the fact that the SimOS uses Alpha 21264 as the bases for SMT features implementation. However, the Alpha architecture, though very well suited to work on any IRIX/MIPS based system, is not easy to acquire and accommodate. For this reason implementation of SMT on the MIPS R10000 has been chosen. In addition to the difficulties with Alpha 21264 mentioned above, the decision to implement SMT on the MIPS R10000 was based upon other reasons, such as:

1) Comparably large selection of documentation materials available for R10000;
2) Compatibility of R10000 with Topsy;
3) A discovery of a very well-structured R4000/Topsy simulator (see previous section);

Even considering all the advances of SMT, there are still some improvements to the underlying hardware needed to be added in order for SMT to be implemented on a single CPU architecture. These improvements were as follows:

1) Duplication of the register file, program counter, subroutine stack and internal processor registers of a superscalar so that the state of multiple threads (contexts) can be held;
2) Implementation of the per-context mechanisms for pipeline flushing, instruction retirement, subroutine returns prediction, and trapping.
2.2.1. Problems associated with basic SimOS/Topsy environment

There is the number of problems that need to be resolved in order to switch from the basic SimOS/Topsy environment to the SMT SimOS/Topsy environment:

1) Initial MXS CPU model is in non-working state
2) The MXS CPU model is not integrated into the basic SimOS/Topsy environment
3) The basic SimOS/Topsy environment does not provide full support for instruction and data caches: the instruction and data caches cannot be loaded with the appropriate segments of Topsy kernel
4) The Topsy kernel requires installation of binary utilities and MIPS cross-compiler

2.2.2. SMT SimOS/Topsy environment

Figure 2 shows the modified SimOS/Topsy integrated environment.

![Figure 2. SMT SimOS/Topsy environment](image)

Upon completion of the compilation and linking processes, the output file, topsy.ecoff, is placed in the root SimOS directory. The modified SimOS simulator includes the implementation of primary and secondary caches and the implementation of the vector of virtual machines. Each of the virtual machines can read instructions and data from instruction and data caches. Both the instructions and data can then be loaded into the vector of CPUs (maximum number of CPUs associated with each machine is 32) and executed by each of the CPUs. Both cache and CPU models are configurable. Even though there are few cache and CPU models available in original SimOS, this work utilizes the modified MXS
CPU model with added SMT capabilities and 2Level cache model of the original SimOS. The user specifies the intended CPU type from the command line. The type and configuration of the instruction cache is specified in the simulator startup configuration file, initSimos. For the list of the main configuration parameters refer to Table. The complete list of the configuration parameters can be found in [31]. The subsequent section provides more details on the work cycle of the simulator.

Table 1. Main configuration parameters of SMT SimOS/Topsy environment

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU.Count</td>
<td>1</td>
<td>Number of CPUs supported by a single virtual machine</td>
</tr>
<tr>
<td>CPU.ISA</td>
<td>MIPS</td>
<td>Instruction set architecture</td>
</tr>
<tr>
<td>CPU.Model</td>
<td>MIPSY</td>
<td>This parameter identifies the type of simulator</td>
</tr>
<tr>
<td>CACHE.Model</td>
<td>2Level</td>
<td>2Level (primary, secondary) caches enabled.</td>
</tr>
<tr>
<td>CACHE.2Level.ISize</td>
<td>1M</td>
<td>1st Level Instruction cache size</td>
</tr>
<tr>
<td>CACHE.2Level.ILine</td>
<td>64</td>
<td>1st Level Instruction cache line size</td>
</tr>
<tr>
<td>TLB.Organization</td>
<td>R10000</td>
<td>MXS TLB organization</td>
</tr>
</tbody>
</table>
2.2.3. Work cycle of the SMT SimOS/Topsy environment

The detailed organization of the SMT SimOS/Topsy integrated environment is shown in Figure 3.

The user starts simulator only if the following steps have been completed:

1. SMT Topsy kernel is built using the MIPSY gcc cross-compiler
2. SMT MXS SimOS/Topsy integrated environment is built using Intel gcc compiler, and the -DSMT option is present in the Makefile of cpus/mxs subdirectory of the simulator
3. The parameters listed in Table 2 are added to initsimos configuration file
The user then enters the following command line in the simos root directory:

```
./simos -X
```

where `-X` option indicates that the simulator utilizes the MXS CPU model. Combined with `CPU.Model` option, this command line option completely defines the simulated environment (MIPSY simulator/SMT MXS CPU).

According to [32], `topsy.ecoff` file contains the following segments: `.text` – text segment of the kernel (instructions), `.data`, `.rdata` – data segments of the kernel (variables and constants). After `main()` function completes the initialization of the simulator, the simulator passes the control to the `LoadImage` function. The `LoadImage` function is responsible for: 1) extraction of the `.text`, `.data`, `.rdata` segments from the `topsy.ecoff` file; 2) loading the instruction cache with the `.text` segment data, and the data cache with the `.data`, `.rdata` segments data.

After instruction and data caches are successfully loaded with the appropriate data, the virtual machine then initializes the vector of defined CPU models (in this case, the vector of the SMT MXS CPUs) and calls the `ms_cycle_once` function. The `ms_cycle_once` function implements a single cycle of execution of a single SMT MXS CPU. The `ms_cycle_once` function calls `ms_fetch` function, which implements the SMT MXS fetching unit and is one of the main emphases of this work.

In addition, the "SMT MXS Fetching Unit" section provides more details on the implementation of the fetching unit and its processing. The “Results” section provides the results of simulation of the none-SMT and SMT MXS fetching units and compares them.
2.3. Annotations vs. Performance Counters

According to SimOS documentation, annotations are mechanisms for running arbitrary TCL scripts throughout the execution of the simulation. Annotations are non-intrusive mechanisms. It means that their execution does not necessarily need to have any affect on the behavior of the workload and operating system under simulation. Annotations are triggered by any number of hardware events, including, but not limited to execution of a specific address, loads or stores to specific data locations, device interrupts, and others. In addition to predefined annotation types, such as discussed above, it is also possible to create user-defined types of annotations. These are typically used to notify scripts of the execution of some higher-level event such as the creation of a new process, entering idle loops, or executing a particular procedure in an application. The simulation uses the annotation command in both instances: to set annotations and to create new annotation types for others to use. Currently, SimOS supports the following annotation types:

- **Pc** – triggers when the PC equals a certain value and fires after all exceptions caused by the instruction’s execution and all cache misses have completed;
- **Load** – this annotation triggers on loads to a particular address;
- **Store** – triggers on all stores to a particular address;
- **Cycle** – triggers when the cycle count reaches the predefined value;
- **Exc** – triggers on exception of a particular type;
- **Inst** – triggers when the instruction of a particular type is executed;
- **Simos** – triggers on specific SimOS events;
- **Scache** – triggers on second level cache misses of a particular type or any type if the type is not specified.

For more details on the format of annotation command please refer to SimOS documentation [26].

The annotations approach has its advantages and disadvantages. Among the advantages, annotations provide flexibility and low execution time overhead. One of the disadvantages of using the annotation approach is a necessity to have multiple scripts to address the different types of annotations, inclusion of TCL libraries into the simulation software, and exporting TCL symbols into the software. Among the problems associated with
the annotation approach there are issues related to installation and initialization of the software related to the specific TCL problems and versions incompatibilities, and the necessity to provide special support of TCL in the software.

Taking into consideration the issues described above, a different approach has been taken, namely performance counters. The performance counters implemented in this work somewhat mimic the Microsoft approach. Every function in simmxs folder writes the data in the series of # - delimited rows, where every number in the row represents a particular measured value of a particular matrix. For example, if some data measurements were written by function X for data series C1 and C2, and the following table represents the actual measured data:

**Table 2. Viewing data with RIT Viewer application**

<table>
<thead>
<tr>
<th>Series C1</th>
<th>Series C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Then the file created by the function X will look like follows:

1 2 #
2 3 #.

A special Windows application RIT_viewer was developed to visualize the data.

Figure 4 demonstrates how RIT_viewer interprits the data from the listed above file.

![RIT Viewer data interpretation](image)

Figure 4. Viewing data with RIT Viewer application
The idea is that the function returns some data that is only meaningful to the interpreter. That allows having a single display for any number of totally different events. Among other advantages, this approach introduces a truly non-intrusive coding style; there is no need to support any extra-libraries; since counters are implemented in C, any environment (including, but not limited to, TCL) may be used for display. For example, the Windows 2000 performance proxy was used to monitor the simulation running on the UNIX box remotely. The user now can use any of these two methods.
Chapter 3. Developing SMT MXS CPU model

- **Section 3.1** contains general specification of the SMT MXS CPU model.
- **Section 3.2** provides detailed description of the SMT MXS instruction set.
- **Section 3.3** explains the organization of the SMT MXS pipeline.
- **Section 3.4** provides brief discussion of the SMT caches, including primary instruction, primary data and secondary caches. This section also describes cache algorithms and virtual-to-physical address translation process. A brief discussion of the operating modes of the SMT MXS CPU model is also provided in this section.
- **Section 3.5** provides detailed description of the fetching unit of the basic MXS CPU model; it then turns to SMT MXS fetching unit and discusses a few problems related to SMT fetching process, as well as possible enhancements to the SMT MXS fetching unit.
- **Section 3.6** introduces the concept of register renaming. The section also demonstrates a few techniques of register renaming that are implemented in the SMT MXS CPU model.
- **Section 3.7** explains the organization of the SMT MXS issue queues.
- **Section 3.8** contains background information on branch prediction policies.
- **Section 3.9** describes the SMP MXS branch prediction unit.
- **Section 3.10** introduces a new dynamic branch prediction policy.
- **Section 3.11** illustrates exception handling techniques implemented in the SMT MXS CPU model.
- **Section 3.12** completes the chapter with the discussion of the SMT changes to MXS Coprocessor 0.
3.1. General Description

Taking a closer look at the SMT shows that the SMT MXS architecture can be categorized as MIPS V (provided that a typical MIPS architecture has been extended four times and the basic R10000 architecture belongs to the MIPS IV category). The following describes basic CPU architecture information, as well as specific SMT information. The detailed description of the basic architecture is contained in [r10000 us. Man]. The specific SMT information is referenced according to the list of sources. The SMT MIPS processor is a single-chip superscalar\(^1\) RISC microprocessor that uses MIPS ANDES\(^2\) architecture and has the following major features [5]:

1) It specifically implements the 64-bit MIPS IV instruction set architecture (ISA);
2) It can decode eight instructions each pipeline cycle, appending them to one of three 32-entry floating point and integer instruction queues;
3) It has ten execution pipelines with the maximum depth of 9 stages connected to separate 6 integer (including 4 Load/Store and 2 Synchronization units) and 4 floating point units;
4) It uses dynamic instruction scheduling\(^3\) and out-of-order execution\(^4\);
5) It uses speculative instruction issue (also termed “speculative branching”\(^5\));
6) It uses non-blocking caches\(^6\);
7) It has separate on-chip 128-Kbyte primary instruction and data caches;
8) It has individually-optimized secondary cache and System interface ports;
9) It has an internal controller for the external secondary cache;
10) It has an internal System interface controller with multiprocessor support.

The table below lists the parameters of the SMT processor and memory system that are suggested to become characteristics of future processors [3]. The sections of this table provide the definitions for each feature, as well as their implementation in the SMT MIPS simulator. All of the listed below parameters are changeable and can be modified by changing configuration and header files of the SMT SIMOS\(^18\) simulator.
Table 3. General specification of SMT microprocessor

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline</td>
<td>9 stages</td>
</tr>
<tr>
<td>Fetch Policy</td>
<td>8 instructions per cycle from up to 2 contexts (the 2,8 scheme of [6])</td>
</tr>
<tr>
<td>Functional Units</td>
<td>6 integer (including 4 load/store and 2 synchronization units), 4 floating point units</td>
</tr>
<tr>
<td>Instruction Queues</td>
<td>32-entry integer and floating point queues</td>
</tr>
<tr>
<td>Renaming Registers</td>
<td>100 integer and 100 floating point</td>
</tr>
<tr>
<td>Registry Files</td>
<td>356 registers to support up to 8 threads on 32-bit architecture [6]</td>
</tr>
<tr>
<td>Retirement bandwidth</td>
<td>12 instructions/cycle</td>
</tr>
<tr>
<td>TLB</td>
<td>128-entry ITLB and DTLB</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>McFarling-style, hybrid predictor [8]</td>
</tr>
<tr>
<td>Local Predictor</td>
<td>4K-entry prediction table indexed by 2K-entry history table</td>
</tr>
<tr>
<td>Global Predictor</td>
<td>8K entries, 8K-entry selection table</td>
</tr>
<tr>
<td>Branch Target Buffer</td>
<td>1K entries, 4-way set associative cache hierarchy</td>
</tr>
<tr>
<td>Cache Line Size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>128KB, 2-way set associative, single port, 2 cycles fill penalty</td>
</tr>
<tr>
<td>Data Cache</td>
<td>128KB, 2-way set associative, dual ported (only from CPU, r/w)</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>16MB, direct mapped, 20 cycles latency, fully pipelined (one access per cycle)</td>
</tr>
<tr>
<td>MSHR</td>
<td>32 entries for the L1 cache, 32 entries for the L2 cache</td>
</tr>
<tr>
<td>Store Buffer</td>
<td>32 entries</td>
</tr>
<tr>
<td>L1-L2 Bus</td>
<td>256-bit side, 2 cycles latency</td>
</tr>
<tr>
<td>Memory Bus</td>
<td>128-bit wide, 4 cycles latency</td>
</tr>
<tr>
<td>Physical Memory</td>
<td>128MB, 90 cycles latency fully pipelined</td>
</tr>
</tbody>
</table>
3.2. Instruction Set

Similar to the R4000 architecture, the SMT MIPS instruction set consists of the following instruction types: immediate (I-type), jump (J-type), and register (R-type).

In the SMT MIPS architecture most of the integer instructions have the pipeline latency\(^9\) of 1 cycle, load/store instructions have the pipeline latency of 1 cycle, branch instructions have the pipeline latency of 1 cycle, float operations are executed iteratively and have the pipeline latency of more than 2 cycles. Load and store instructions move data between memory and general registers. They all are of I-type, except for special instructions. Computational instructions perform arithmetic, logical, shift, multiply, and divide operations on values in registers. They include register- R-type, in which both the operands and the result are stored in registers; and immediate- I-type, in which one operand is a 16-bit immediate value, formats. Jump and Branch instructions change the control flow of a program. Jumps are always made to a paged, absolute address formed by combining a 26-bit target address with the high-order bits of the Program Counter (J-type format) or register address (R-type format). Branches have 16-bit offsets relative to the program counter (I-type). Jump and Link instructions save their return address in register 31. Coprocessor instructions perform operations in the coprocessor. Coprocessor load and store operations are I-type. Coprocessor 0 (system coprocessor) instructions perform operations on CP0 registers to control the memory management and exception handling facilities of the processor. Due to their importance to further discussion, their explicit list will be provided in the “Coprocessor 0” section of this work. Special instructions perform system calls and breakpoint operations. These instructions are always R-type. They will not be, however, discussed in this work. Exception instructions cause a jump to the general exception-handling vector. This jump is based upon the result of a comparison. These instructions occur in both R-type (both the operands and the result are registers) and I-type (one operand is a 16-bit immediate value). The SMT architecture also should have additional three types of instructions [4]:

1) Instructions which affect physical contexts (to stop, start, read, or write the PC of a given context, or to access the port on which a context is waiting);

2) Instructions to exchange data between and synchronize two physical contexts;
3) Instructions to exploit physical locks.

To utilize those extensions, [4] suggests implementation of a special functional unit. This functional unit takes care of informing the various stages of the processor of the activation stage of the contexts execution mask. The instructions belonging to active contexts can go through the pipeline to be executed. The suspended contexts do not have access to the pipeline. The instructions already executed (i.e., in the write back or commit states) before the arrival of blocking instruction have the right to complete, while the others are flushed out of the pipeline, exactly as for a branch misprediction. The placement of these three types of instructions in the same unit, according to [4], has been motivated by great similarity in their actions on the ordering of the instructions. Similar to synchronous data exchanges, locks require the ability to begin and terminate the execution of physical contexts. This control affects the instructions in their decode/dispatch, write-back, and commit stages and, therefore, should be centralized. This unit is used in the similar manner as the load/store unit when it has to write a word back into the main memory. The main limitation on these instructions is that they must be executed strictly in non-speculative mode to avoid inconsistent results. For example, if the message-based operating system must deliver a signal from one context to another, and this signal happens to get sent right after the execution of a branch, the message can be delivered only if that branch is taken. Without explicit locking mechanisms in place, provided that the simultaneous execution of instructions is coming from different contexts, there is no guarantee that the message will not be read between the branch and the speculative delivery of the message. If the branch were mispredicted, it would then be necessary to correct the context, which reads the message and invalidates all the instructions since the reading of the message.

To access the context control and communication/lock unit, [4] adds two instructions to the instruction set allowing a context to read from (rport Rd, Rp) and write to a port (wport Rs, Rp). The register Rp holds the number of the port to access. The register Rd holds the value read from the port. The Register Rs holds the value to send to the port. Any of the 32 general-purpose registers may be used for Rp, Rd and Rs. The port numbers are coded on 32 bits, except for few ones that are reserved for special use: data exchanges through ports, for instance, require numbers greater than 0xff. Of course, the instruction set must support encoding the rport and wport instructions in some way. It would greatly simplify the access
to the communication ports and to the locks, the numbers no longer identifying the function but only the particular port or lock. In the SMT MIPS architecture, the execution or the termination of any process may be controlled by the process itself or by a different process. When the request is external, it is very likely that it was generated as the result of a context switch. Similarly, when a context is started by another context it is very possible that a context switch has just occurred (unless it is a result of an exchange of messages between two processes in two different physical contexts or of the release of a lock). The context switches can be controlled by the leaving context. The access (both writing and reading) to another context’s PC and the existence of a shared memory area are sufficient to perform this task. Those two operations are:

1) Reading a particular context’s PC that returns its value and suspends it on the instructions pointed by the PC. The context will then resume from that same instruction address;

2) Writing into a particular context’s PC causes it to resume.

Starting a new process in a context is done by preparing the associated stack and inserting in it the execution context. After that, the address of a minimal loading procedure is written into the PC of the context that is going to execute the process. The process starts and waits for a specified message port. The initiating process then writes to this particular port the stack pointer. The staring context reads this value, loads the execution context into the registers and jumps to the portion of code it should execute. A context switch occurs in the same way. A master process, which decides to suspend another process, writes into the PC of the slave’s context the address of the procedure that moves the context back to the memory. This causes the slave’s registers to be written to the stack and its stack pointer to be sent as a message back to the master. At this point the physical context is ready for use by another process. It is then sufficient to write into the PC the address of the loading subroutine for the incoming process. It first reads a message on the port where the master process writes the value of the appropriate stack pointer. With it, it is possible to retrieve the correct execution context. Once the values of all the registers have been restored, the master can write into PC the address of the instructions executed by the resuming process at the time it was suspended. Therefore, to stop, start, or switch a context, the ability to access another context’s PC and the existence of a data transmission mechanism are sufficient. As
suggested by [4], the contexts exchange the data via a shared piece of memory. The problem, however, is that the cache can be erased before it's requested by the destination context. As [4] explains, the solution that combines distributed caches with a snoopy mechanism that detects when a context is trying to access data which is stored in the cache of another context) also will not work. Even if the unit uses such a cache in write-through mode, resolving the coherency problem, the transfer time of the data significantly increases (now there is whole round is required to transmit a piece of data). Subsequently, the new implementation has to accommodate the following ideas:

1) An ability to read from and write to the individual context's port;
2) A signaling mechanism that informs the context of the action;
3) A blocking mechanism to ensure the data synchronization.

The exchange mechanism that was proposed by [4] and implemented as a part of this work relies on the notion of communication ports. A communication port is a one-word-wide data register, associated with a port number. Reading or writing to a port is blocking (until execution or the corresponding write or read occurs) allowing synchronous exchange of data. The context management unit thus contains an associative table of 6 entries (one entry per user thread). It is useless to have more entries than contexts. This table allows the implementation of these communication ports. The format of the table's entries is described below and shown in Figure 5.

![Port table entry](image)

Figure 5. Port table entry.

Courtesy of Dominik Madoń, Eduardo Sánchez, Stefan Monnier, Computer Science Department, Yale University

Each line contains a field for the number of the port used, a field for the value, two fields for the physical numbers of the contexts, which are using this entry (the sender and the
recipient), and three status flags. This entry format allows a number of contexts to use the same port number, in which case the requests will be satisfied in an arbitrary order. Sending and receiving are almost symmetrical from the point of view of their operation. This structure allows any context to be suspended if it is waiting on a port. The process, when it resumes, will execute the same instruction (send or receive), which will again result in either the exchange of data or another suspension. The context, which owns an entry where two flags (W and D or, R and D) are set to one, will not be stopped or will not be able to change PCs until the re-execution of the communication instruction, which erases all the flags. This mechanism ensures that a process cannot stop a context that has completed a message exchange but has not yet re-executed the rport or wport instruction in order to increment its PC. Synchronous ports are meant for message exchange by t can of cause be used to implement locks. This solution is inconvenient in that it requires auxiliary contexts (since a communication port links two contexts, a sender and a recipient, while locks require only a single process). Moreover, locks should also operate as fast as possible. For these two reasons [4] suggests creating a lock table, whose entries have the special format. These locks allow critical sections to be protected using the access instructions wport and rport of the context control unit. The suspension of a context, when a lock prevents access, uses communication ports as a chained list (queue) of waiting contexts. This method allows to reuse a structure already available and to transfer to memory contexts waiting on a lock. Each entry from the lock table (6 in total) contains a communication port number for the next context wishing to access the critical section, a communication port number for the last context on standby in the queue, the number of the context which has set the lock, and a flag S (set) indicating if the lock is available or busy.
3.3. Pipeline

The processors of MIPS III (R4000) generations had linear pipelines that consisted of the following stages:

- Instruction fetch;
- Instruction decode and dependency;
- Execution;
- Data fetch;
- Data decode and dependency;
- Write back;
- References to register file;
- TLB.

Such pipeline, in theory, could provide a maximum throughput of one instruction per cycle, provided that both memory and data dependencies were successfully resolved. Typical Superscalar architecture, on the other hand, provided shallower pipeline that contained 5 stages. The gain in performance was achieved by letting the fetch of 4 instructions (4-way Superscalar) on every CPU cycle. In this particular research the SMT MIPS architecture represents an 8-way Superscalar with 10 pipelines.

SMT MIPS pipeline supports the following features [6]:

1) Multiple program counters and some mechanism by which the fetch unit selects one each cycle;
2) A separate return stack for each thread for predicting subroutine return destinations;
3) Per-thread instruction retirement, instruction queue flush, and trap mechanisms;
4) A thread id with each branch target buffer entry to avoid predicting phantom branches;
5) A larger register file, to support logical registers for register renaming.

The size of the register file affects the pipeline’s number of stages (extra 2, comparing to R10000) and the scheduling of load-dependent instructions.

The fetching is done from one program counter each cycle. The PC is chosen in round-robin manner from among the threads that do not experience the early I cache miss. For more information on fetching unit see the “Fetching Unit” section of this work.
In order to consider a large (356 registers) size of the register file, the SMT MIPS pipeline has two cycles to read and write registers. This idea is somewhat similar to reservation station: in the first cycle the information is copied to the attached buffer; in the next cycle the information is sent to/from the register file. It increases the pipeline distance between fetch and exec by increasing the branch misprediction penalty by 1 cycle. It also takes an extra cycle to write back results, requiring an extra level of bypass logic. Finally, increasing the distance between queue and exec increases the period during which wrong-path instructions\(^7\) remain in the pipeline after a misprediction is discovered. For more information on fetching unit and fetching policy see the “Fetching Unit” section of this work.

Stage 1. In this stage the processor fetches eight instructions each cycle, independent of their alignment in the instruction cache – except that the processor cannot fetch across a 16-word cache block boundary. These words are then aligned in the 4-word Instruction register. If any instructions were left from the previous decode cycle, they are merged with new words from the instruction cache to fill the Instruction register.

Stages 2-3. In these stages, the eight instructions in the Instruction register are decoded and renamed (in order to eliminate instruction dependencies and provide precise exception handling\(^10\)). As each instruction is renamed, its logical register numbers are compared to determine if any dependencies exist between the eight instructions decoded during this cycle. After the physical register numbers become known, the Physical Register Busy table (busy list) indicates whether or not each operand is valid. The renamed instructions are loaded into integer or floating-point instruction queues. Comparing to SMT MIPS, original R10000 architecture did not expand this process to the next stage, which, in this case, would have been Stage 3. In addition, there is also a problem with SMT MIPS architecture, such as a very slow access to the register file due to the size of the renaming map. If no special precautions are taken in order to deal with this problem, the size of the register file will definitely affect the machine cycle. Therefore, at the end of the second stage the register values are copied into the buffers that are close to the execution units, and then the register file is updated at the beginning of the third stage of the pipeline.

A scheduling decision to read the next program counter is made by the CPU and based on the following factors [8]:

35
1) The thread ID of the process, which is associated with the next PC candidate that is not marked as “locked” (the “locked” attribute is maintained by the CPU for every thread ID within a particular context). In other words, the next thread must be scheduled for execution by the OS scheduler;
2) Whether the time slice for the previous counter has expired or the thread associated with the previous PC has become blocked or was killed as the result of the exception;
3) Whether the previous thread has finished execution.

It is usually set by the CPU in the event of any exception or by the kernel if either the external request or system call occurs. Each branch instruction is indexed with the thread ID. It is necessary to determine which program counter must be updated. Up to 4 branches can execute per cycle. The branch unit determines the next address for the Program Counter. The CPU controls the updates to the program counters by generating a special exception. Every program counter is assigned to its own register, therefore, unlike any other exception, the time exception does not lead to the WAW hazard. For more information on exception handling see “Exceptions Handling” subsection of the SMT CPU model. If a branch is taken and then reversed, the branch resume cache provides the instructions to be decoded during the next cycle.

Stage 4. In this stage, decoded instructions are written into the queues. On the same token, stage 3 is also the start of each of the ten execution pipelines.

Stages 5-7. In stages 5-7, instructions are executed in the various functional units. These units and their execution process are described as following:

1) Floating-Point Multiplier (3-stage Pipeline) – Single- or double precision multiply and conditional move operations are executed in this unit with a 2-cycle latency and a 1-cycle repeat rate. The multiplication is completed during the first two cycles; the third cycle is used to pack and transfer the result;
2) Floating-Point Divide and Square-Root – can executed in parallel, share their issue and completion logic with the floating-point multiplier;
3) Floating-Point Adder (3-stage Pipeline) – single- or double-precision add, subtract, compare, or convert operations are executed with a 2-cycle latency and a 1-cycle repeat rate. Although a final result is not calculated until the third pipeline stage, internal bypass paths set a 2-cycle latency for dependent add or multiply instructions (forwarding logic).
4) Integer ALU1 (1-stage Pipeline) – Integer add, subtract, shift, and logic operations are executed with a 1-cycle latency and a 1-cycle repeat rate. This ALU also verifies predictions made for branches that are conditional on integer register values;

5) Integer ALU2 (1-stage Pipeline) – integer add, subtract, and logic operations are executed with a 1-cycle latency and a 1-cycle repeat rate. Integer multiply and divide operations take more than one cycle.

The list of independent instructions for SMT MIPS matches the one for R10000. The hardware can fetch up to eight instructions on every cycle and issue up to ten instructions on every cycle. Every cycle the hardware would select one instruction from each of the columns from the Table2. Floating-point divide, floating-point square root, integer multiply and integer divide cannot be started on each cycle [5].

Table 4. Instructions compatibility in SMT microprocessor

<table>
<thead>
<tr>
<th>Column A</th>
<th>Column B</th>
<th>Column C</th>
<th>Column D</th>
<th>Column E</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPadd</td>
<td>FPmul</td>
<td>FPload</td>
<td>add/sub</td>
<td>Add/sub</td>
</tr>
<tr>
<td></td>
<td>FPdiv</td>
<td>FPstore</td>
<td>Shift</td>
<td>Mul</td>
</tr>
<tr>
<td></td>
<td>FPsqrt</td>
<td>Load</td>
<td>Branch</td>
<td>Div</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Store</td>
<td>Logical</td>
<td>Logical</td>
</tr>
</tbody>
</table>

Stages 8-9. At this stages up to 8 instructions graduate and, consequently, are written back to the main memory. Due to the size of the register file, the actual write also takes two stages. At this point, all the registers assigned to the graduating instruction are freed and put back in the free list, and an appropriate page in the primary memory is marked as "dirty". Since SMT MIPS implements the write back policy, the actual write to the secondary cache and persistent storage is postponed.
3.4. Caches

This section discusses instruction and data caches of the SMT MXS CPU model.

3.4.1. Primary Instruction Cache

The SMT MIPS architecture inherits its caches from the original MIPS R10000 architecture. For more information see [5]. Primary Instruction cache is a 128 Kbytes-entry and two-way associative. The line size of the primary instruction cache is 16 words with LRU\textsuperscript{123} replacement algorithm. The cache is indexed with a virtual address and tagged with physical address. At any time the primary instruction cache may be in either Valid or Invalid state. The cache can change its state as the result of the following events:
1) A primary instruction cache read miss;
2) Subset property enforcement;
3) Any CACHE instruction execution;
4) External invalidate requests.

3.4.2. Primary Data Cache

Primary Data Cache is a 128 Kbytes-entry and two-way associative. The line size of the primary data cache is 8 words with LRU replacement algorithm. The cache writes back to the secondary data cache, and the secondary cache, in turn, writes back to the main memory under control of the system interface. The data cache is indexed with a virtual address and tagged with a physical address and can be in either one of the following states (see [5] for detailed descriptions of the stages):
1) Invalid;
2) CleanExclusive;
3) DirtyExclusive;
4) Shared.

The following events can change the state of the primary data cache:
1) Primary data cache read/write miss;
2) Primary data cache write hit;
3) Subset enforcement;
4) A CACHE instruction;
5) External intervention shared request;
6) Intervention exclusive request;
7) Invalidate request.

3.4.3. Secondary Cache

Since the structure and behavior of the secondary cache for R10000 and SMT MIPS are the same, the description of the secondary cache will be omitted in this work and the reader is referred to [5].

3.4.4. Cache Algorithms

The SMT MIPS processor uses the same set of cache algorithms as the R10000. These algorithms are identified using:
1) Per-address basis by 3-bit cache algorithm field in the TLB, where the values are encoded into EntryLo0 and EntryLo1 registers;
2) The 3-bit K0 field of the CP0 Config register for non-cacheable kseg0; and
3) The 61:59 bits of the xkphys non-cacheable space.

Regardless of which method is used, those values remain fixed and can be found in [5]. For further discussion, though, short descriptions of those algorithms should be provided:

*Uncached* – loads and stores bypass the primary and secondary caches. They are issued directly to the System interface using processor double/single/partial-word read or write requests.

*Cacheable Noncoherent* – load and store secondary cache misses result in processor noncoherent block read requests.

*Cacheable Coherent Exclusive* – load and store secondary cache misses result in processor coherent block read exclusive requests.
Cacheable Coherent Exclusive on Write – load and store secondary cache misses result in processor coherent block read shared requests.

Uncached Accelerated – this algorithm allows the kernel to mark the TLB entries for certain regions of the physical address space, or certain blocks of data, as uncached while signaling to the hardware that data movement optimizations are permissible.

3.4.5. Virtual Address Translation and Pages

The processor accepts a program’s addresses in two forms: physical or virtual. The virtual form is essential for multitasking computer systems, since it allows the operating system to load each program anywhere independent of the logical addresses used by the program. Also, the virtual translation implements a memory protection schema, since each program can use the entire memory space without interfering with the memory used by another programs (including operating system).

Operating modes:
1) User mode;
2) Supervisor mode;
3) Kernel mode.

These modes will be described along with associated translation techniques, in the following paragraphs. The following table represents the CPU settings for each mode.

Table 5. Operational modes of SMT MIPS microprocessor

<table>
<thead>
<tr>
<th>XX 31</th>
<th>KX 7</th>
<th>SX 6</th>
<th>UX 5</th>
<th>KSU 4:3</th>
<th>ERL 2</th>
<th>EXL 1</th>
<th>Description</th>
<th>ISA III</th>
<th>ISA IV</th>
<th>Addr.mode 32/64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>User mode</td>
<td>No</td>
<td>No</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
<td>No</td>
<td>Yes</td>
<td>32</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
<td>Yes</td>
<td>No</td>
<td>64</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>64</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>Supervisor mode</td>
<td>No</td>
<td>Yes</td>
<td>32</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>64</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>Kernel mode</td>
<td>Yes</td>
<td>Yes</td>
<td>32</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>64</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>Exception Level</td>
<td>Yes</td>
<td>Yes</td>
<td>32</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>64</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>X</td>
<td></td>
<td>Error Level</td>
<td>Yes</td>
<td>Yes</td>
<td>32</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>64</td>
</tr>
</tbody>
</table>
Since SMT Topsy operates in the same modes, it is not considered reasonable to provide the detailed description of those, as well as the description of the basic R4000 TLB architecture.

3.5. Fetching Unit of SMT MXS CPU model

This section provides the description of basic MXS fetching unit, SMT MXS fetching unit and their implementation details.

3.5.1 Basic MXS Fetching Unit

As it was previously discussed in “Developing SMT SimOS/Topsy environment” section of this work, the basic MXS fetching unit is implemented in the ms_fetch function. This function is invoked by the ms_cycle_once function that simulates a single work cycle of the MXS CPU model.

The basic MXS fetching unit is implemented in the ms_fetch function. The function begins by updating the CPU state. After that it uses the current PC value to determine the index tag and calls TranslateVirtual() function to identify the physical address associated with the next available instruction in the instruction cache. This physical address represents the tag of the line of the instruction cache. Once the cache line is identified and validated, the ReadICache() function reads the next available instruction from the instruction cache. Then the ms_fetch function searches all active threads in order to identify if there are any instructions (their count varies from 0 to the maximum number of instructions per fetch FETCH_WIDTH). Then the preliminary instruction check is made, and the function continues its execution by providing various checks on the instruction and inserting the instruction into one of the issue queues.

The current value of the PC counter is loaded into PC register of the MXS simulator by Topsy for currently scheduled process. The scheduling decision is made by the MXS
scheduler. For more details on Topsy scheduling process see "Basic Topsy Scheduler" section of this work.

3.5.2 SMT MXS Fetching Unit

When the basic SimOS environment is compiled with -DSMT option, the basic MXS fetching unit is transformed into the SMT MXS fetching unit. The SMT MXS fetching unit is implemented on top of the ms_fetch function in ms_cycle_once function. This section provides a short discussion of the SMT MXS fetching process and differences between none-SMT and SMT fetching processes. Then the section introduces the problem associated with SMT fetching in SMT MXS simulator. The solution to the problem, along with the results of the SMT MXS fetching, is then described in the "Results" section of this work.

As it was described in the "Developing SMT SimOS/Topsy environment" section of this work, the SMT MXS fetching unit fetches the instructions from several processes, scheduled for execution at a particular time slice, simultaneously, based on the state of the simulator, the state of a particular process (exceptions, type of the instructions) and some additional restrictions placed on the process of simulation (for details on additional restriction to the process of simulation see the "Enhanced SMT MXS Fetching Unit" section of this work). The next available instruction of each process is read by the SMT MXS fetching unit from the instruction cache. As it was explained in the previous section of this work, the address of the next available instruction of the process is identified, using the PC register of the simulator. The basic difference between the SMT MXS and the basic MXS fetching processes is that, unlike in the basic one, there is the number of PC counters available to the SMT MXS fetching unit on any particular cycle of execution (a call to ms_cycle_once function). Each PC counter has an associated PC counter register in the SMT MXS SimOS. Each PC counter register has an associated "slot" in the RUNNING queue of the SMT Topsy. When a particular process is scheduled for execution during the next time slice, it loads the current value of its PC count into the associated PC counter register, based on the "slot" number of the scheduled process in the RUNNING queue. When a time slice expires,
or for some other reasons, the process changes its state to either BLOCKED or READY. In any case, the process loads 0 into the associated PC counter. By loading 0 into the PC register, the process lets the SMT MXS fetching unit know that no instructions can be fetched from the associated “slot” on this cycle of fetching. For more details on the SMT Topsy scheduling process see the “SMT Topsy Scheduler” section of this work.

The main problem associated with the SMT MXS fetching process is that each of the PC counter registers is loaded dynamically by SMT Topsy (a call to the RestoreContext function). However, since the ms_execute function is not functional, none of the PC counter registers can be dynamically re-initialized. The solution to this problem is discussed in the “Results” section of this work.

3.5.3 Enhanced SMT MXS Fetching Unit

As was previously noted, the utilized 2, 8 fetching schema will be discussed in more details in the “SMT Topsy” section of this work. However, the factors that affect the instruction fetch and the ways of implementation of a non-blocking (or rather “minimally blocked”) fetching unit seem reasonable to be discussed here. This section will be concluded with the discussion of the sim Ms_fetch function. This function has been developed as a part of the SMT enhancements to the standard MXS architecture, which were implemented in this work. According to [6], there are two factors that affect the CPU choice of the next fetched thread.

The first factor is the one where there is a probability of the thread taking the wrong path of execution after an early branch misprediction. In this case all the instructions on the mispredicted path should be removed from the pipeline and their operand registers should be returned to the register file.

The second factor is represented by the time that a particular instruction spends in the instruction queue before being issued. One approach to the problem is to feed the queue with the instructions that will spend the least time in it. If there would be too many instructions fetched in the instruction queue, the queue will be eventually filled with unusable instructions and clog the IQ (the term that has been introduced by professor Tullsen). For instance, he suggested a few fetch policies, each of which attempts to improve on the round-robin priority
policy using feedback from other parts of the processor. One of the strategies deals with the wrong-path fetching, whereas the others attempt to eliminate IQ clog. The following are the descriptions of each strategy suggested by professor Tullsen and the ways they were implemented in this work [17].

BCOUNT - the CPU gives the highest priority to those threads that are least likely to be on a wrong path. In this work it is done by counting brand instruction that are in the decode stage, the rename stage, and the instruction queues, favoring those with the fewest unresolved branches.

MISCOUNT - the long memory latency is an important cause of the IQ clog (horizontal waste). Long memory latency causes dependent instructions to back up in the IQ till the load is complete, filling the IQ with blocked instructions from one thread. This policy prevents that by giving priority to those threads that have the fewest outstanding data cache misses.

ICOUNT - the priority is given to the threads with fewer instructions in decode, rename stages and in the queues. This strategy is the most general solution to both problems, since it:
Prevents any one thread from filling the IQ;
Gives the highest priority to threads that are moving instructions through the IQ most efficiently;
Provides a more even mix of instructions from the available threads, increasing the parallelism in the queues.

The interested user can experiment with the three listed above fetching policies by choosing the FStrategy parameter from the f_strategy enumerator (see sim_ms_fetch.h file for details).

IQPOSN - this policy behaves much like ICOUNT, attempting to minimize IQ clog and bias toward efficient threads. It gives lowest priority to those threads with instructions closest to the head of either the integer or floating-point instruction queues (the queues are FIFO). Threads with the oldest instructions will be the main subjects to the IQ clog. The advantage of this strategy is that it does not require the counter per thread.

As Tullsen showed in [6], the fetching from multiple threads combined with sophisticated fetch heuristics might significantly increase fetch throughput and efficiency.
Therefore, if the fetch unit becomes unusable during any machine cycle, the instruction penalty increases dramatically. The following two schemes prevent the fetch unit from IQ-full conditions (the BIGQ modification) and I cache misses (the ITAG strategy). The BIGQ assertion simply puts the 32-entry limit on the length of all the instruction queues. The ITAG algorithm, however, requires some further explanations. For more implementation details please refer to the ITAG function.

The main idea of the ITAG is to allow the I-cache tag lookup a cycle on the early stage. The ITAG function returns the array of those thread IDs that experience the I-miss (stalled on TLB or cache refill). Only non-missing threads are chosen for fetch. Since there must be an address fetch cycle early, another pre-fetch stage must be added to the pipeline, increasing the pipeline penalties. Analysis showed that if the number of I-cache tag ports were increased, this problem would be effectively eliminated.

The previous discussion is also applicable to the issue stage, where the depth of a particular instruction in the instruction queue is no longer the best indicator of when the instruction can be issued. The InstCompliesWithFetchPolicy function has been implemented in order to be able to examine different fetching strategies suggested by Tullsen. The implemented policies are OLDEST_FIRST (the default issue algorithm), OPT_LAST and SPEC_LAST, which only issue optimistic and speculative instructions (more specifically, any instruction behind a branch from the same thread in the instruction queue), respectively, after all others have been issued, and BRANCH_FIRST, which issues branches as early as possible in order to identify mispredicted branches quickly. The default fetching algorithm for every issue policy is ICOUNT2.8 and should not be changed [17].

As it has been previously discussed, one of the main advantages of the SMT CPU is its ability to make smart choices about contexts, from which it fetches the instructions at any particular time. There are many policies that define which threads must be chosen by the processor at any instant time. However, very few sources provide actual algorithms to allow this capability. In SMT MXS simulator, a true SMT fetching unit has been developed in the course of this work that allows:

- Fetching from several different threads at any time of execution;
- Switching the fetch policy to study the fetching behavior of different workloads among different fetching policies.
The SMT MXS support for fetching can be found in file smt_ms_fetch.c. The SMT MXS simulator supports the following fetching unit enumerators and functions:

- **FPOLICY** – enumerated data type that identifies possible fetching policies. For more information on fetching policies see the previous section of this work;
- **CurrentFetchPolicy**. This global variable of type FPOLICY specifies current global fetching policy of the SMT simulator;
- **InstCompliesWithFetchPolicy**. This function returns true if the current instruction is a candidate to be fetched into the CPU, complies with the current CPU fetching policy. The following paragraphs will on this function;
- **InstInWindow**. This function returns the total number of instructions currently in execution window;
- **BrInWindow**. Like the previous one, this function returns the total number of branches that are currently in the execution window;
- **ExecuteCDI**. The function provides support for the OS CDI instruction. This instruction allows switching dead contexts from the CPU and returning associated renamed registers into the free list of available registers.

For more details see the “Register Renaming and Special Techniques” section of this work.

An ability to switch fetching policies requires an addition of new members to the THREAD (s_threads) structure. This structure represents an active thread of execution within the simulator and can be found in file sim_ms.h. The following members were added to the structure to support a wide variety of fetching policies:

- **B_DS_CNT** – the number of branches in the decode stage;
- **B_RS_CNT** - the number of branches in the rename stage;
- **ODCM_CNT** – the number of outstanding data cache misses;
- **OICM_CNT** – the number of outstanding instruction cache misses;
- **I_DS_CNT** – the total number of instructions in the decode stage;
- **I_RS_CNT** – the total number of instructions in the rename stage;

At this point it is reasonable to proceed with the discussion on the sim_ms_fetch function.

The function begins by updating the CPU state. Then it searches all active threads in order to identify if there are any instructions (their count varies from 0 to the maximum number of instructions per fetch FETCH_WIDTH). Then the preliminary instruction check
is made. If the instruction is CDI, the function invalidates the renaming map by making a call to the InvalidateMap function and continues with the next available instruction. Otherwise, the function continues its execution by checking whether the instruction is a branch. If this condition holds true, the B_DS_CNT member of the thread is updated. The function then passes control to the InstCompliesWithFetchPolicy function.

The InstCompliesWithFetchPolicy function identifies the thread that has the minimal value of one of its counters. The choice of the counter value is based upon the value of the CurrentFetchPolicy constant. Once the thread has been identified, the function then compares its index with the index of the current thread in focus. If those indices match, the function returns true, otherwise it return false. The control is then passed back to the sim_ms_fetch function.

The sim_ms_fetch function then continues its processing, which is very complex and is not a subject for a detailed description in this work. It is only worth mentioning that the sim_ms_fetch function utilizes every aspect of the SMT simulator development.

### 3.6. Register Renaming Unit and Special Techniques

The R10000 architecture provides a single 32-entry register file. There is only one context that acquires the file. Therefore, there is no need for any register sharing, and there is no need to provide any special techniques to increase the architectural performance. The following discussion focuses on supporting the effective sharing of registers in the SMT processor, using register renaming to permit multiple threads to share a single global register file. For instance, a single thread with the high renaming demand can benefit when other threads have low register demands. The problem with any existing hardware renaming techniques is that none of them can fully exploit all the advantages of the sharing register file. In particular, while existing hardware is effective at allocating physical registers, it has only limited ability to identify register deallocation points; therefore hardware must free registers conservatively, possibly wasting registers that could be better utilized. The [7] proposes the software support for two types of dead registers:
1) Registers allocated to idle hardware contexts;
2) Registers in active contexts whose last use has already retired.

In the first case, when a thread terminates execution on the SMT architecture, its hardware context becomes idle if no threads are ready to run. While those registers that are allocated to the idle thread are dead, they are not freed, because hardware register deallocation only occurs when registers in a new, active thread are mapped. This causes a potentially shared SMT register file to behave like a partitioned collection of per-thread registers. In the second case, [7] presents five compiler mechanisms that allow for the communication of the last-use information to the processor, so that the renaming hardware can deallocate registers more aggressively. Without this information, the hardware must deallocate registers only after they are redefined.

The techniques proposed by [7] assumes the fetching unit that implements ICOUNT2,8 policy (for more details see the “Fetching Unit” section). After fetching, instructions are decoded, their registers are renamed, and they are inserted into either the integer or floating-point instruction queues (one queue is totally eliminated from the architecture; see R10000 pipeline for more details). When their operands become available, instructions (from any thread) are issued to the functional units that execute them. Finally, the instructions are retired in order per thread. The register renaming hardware is responsible for three main functions:
1) Physical register allocation;
2) Register operand renaming;
3) Register deallocation.

Physical register allocation occurs on demand. When an instruction requests an architectural register, a mapping is created from this register to the next available physical register and is entered into to mapping table. The physical register is removed from the free list and is added to the busy registers list. If no registers are available, instruction fetching stalls. To rename a register operand, the renaming hardware locates its architectural-to-physical mapping in the mapping table and aliases it to its physical number. As was mentioned before, the hardware handles the allocation and renaming effectively, but fails to support effective deallocation. The main reason for this is that the hardware cannot effectively identify the last use of the graduated instruction because it has no knowledge of
register lifetimes. There are two possibilities that are exploited by a few architectural choices. One is a partitioned file for one thread, and a thread only accesses registers from its own context. The other possibility, the SMT renaming policy, utilizes the fully shared registers, because the register file is structured as a single pool of physical registers and holds the state of all resident threads. The SMT’s register renaming hardware is essentially an extension of the register-mapping scheme to multiple contexts. Threads name architectural registers from their own context, and the renaming hardware maps these thread-private, architectural registers to the pool of thread-independent physical registers. Register renaming thus provides a transparent mechanism for sharing the register pool. Indeed, the best utilization of the SMT processor is achieved when all hardware contexts are busy. However, some contexts sometimes might become idle. In order to maximize performance, if any thread within a context should become idle, no physical registers should be allocated to such the thread; instead, the active threads should share all physical registers. However, with existing schemes, when a thread either terminates or is stalled on the exception, its architectural registers remain allocated in the processor unit they are redefined by a new thread executing in the context, or in the worst case scenario, when the context switch occurs. Consequently, the fully shared register file behaves more like a partitioned file, significantly lowering the performance of the SMT architecture. The following paragraphs will discuss two general solutions to the problem of increasing the SMT architectural performance:

1) Operating system support;
2) Compiler support.

Providing the function, which implements and manages the register-renaming unit of the SMT MIPS, will conclude this section.

### 3.6.1. Operating System Support for Register Renaming

After a thread terminates, the operating system decides what should be scheduled on the newly available hardware context. There are three options, each of which has a different implication for register deallocation:
1) Switching to an idle thread: the code switches to a different thread within a context. As its instructions retire, physical registers used by the terminated thread are deallocated;

2) Switching to an active thread – physical registers for a new thread’s architectural registers are normally allocated when it begins execution (either a stand-alone thread or a thread within a context switch in the SMT CPU). A more efficient scheme should free the terminated threads’ physical registers, allocating physical registers to the new thread (or a context) on demand. In the SMT case, unallocated physical registers would become available to other threads within the context;

3) Idle contexts – if there are no new threads to run, the context will become idle (it’s impossible in case of Topsy, since there is always at least three threads ready to run at any given moment of time.

Those are the Memory Manager, the Thread Manager and the Idle User Thread. For more details see “SMT Topsy” section of this work). The terminated threads’ physical registers could be deallocated, to that they become available to active threads. All three scenarios provide an opportunity to deallocate terminated threads’ physical registers early. The [7] proposes a privileged, context deallocation instruction (CDI) that triggers physical register deallocation for a thread. The operating system scheduler (for more details see “SMT Scheduler”) executes the instruction in the context of the terminated thread. In response, the renaming hardware would free the terminating thread’s physical registers when the instruction retires.

Three tasks must be performed to handle the context deallocation instruction:

1) Creating a new mapping table;

2) Invalidating the thread (context) register mappings;

3) And returning the registers to the free list.

When the CDI instruction enters pipeline, the current mapping table is saved and a new mapping table is created, which has no valid entries; the saved mapping table identifies the physical registers that should be deallocated, while the new table will hold subsequent register mappings. Once the CDI retires, the saved map is traversed, and all mapped physical registers are returned to the free list. Finally, all entries in the saved map are invalidated. If the CDI is executed on a wrong-path and consequently gets deleted, both the new and saved
mapping tables are thrown away. In the existing simulator branch logic combined with the renaming utility InvalidateMap function can be used to support this task: when a branch enters the pipeline, a copy of the mapping table is created; when the branch is resolved, one of the mapping tables is invalidated, depending on whether the speculation was correct. If instructions must be removed, the renaming logic traverses the active list to determine which physical registers should be returned to the free list. Although the CDI adds a small amount of logic to existing renaming hardware, it allows the SMT register file to behave as true fully shared renaming unit, instead of the partitioned file by de-allocating registers more promptly.

3.6.2. Compiler Support for Register Renaming

There are three factors that motivate the need for improved register deallocation:
1) How often physical registers are unavailable;
2) How many registers are dead each cycle;
3) Dead-register distance\(^{14}\).

Register unavailability is the percentage of total execution cycles in which the processor runs out of physical registers (causing fetch stalls); it can be a measure of the severity of the problem caused by current hardware register de-allocation mechanisms. The average number of dead registers each cycle indicates how many physical registers can be reused, and, therefore, provide the potential for a compile-based solution. Dead-register distance measures the average number of cycles between the completion of an instruction that last uses a register and that register's de-allocation; it is a rough estimate of the likely performance gain of a solution.

Using dataflow analysis, the compiler can identify the last use of a register value. The [7] introduces three alternatives for communicating last-use information to the renaming hardware:
1) **Free Register Bit** – the last-use information is transmitted to the hardware via dedicated instruction bits, with the dual benefits of immediately identifying last uses and requiring no instruction overhead. This solution was not implemented in this function, because most instruction sets do not have two unused bits. However, it has
been realized that this is the most effective solution that puts an upper limit on performance improvements that can be attained with the compiler’s static last-use information. In general to simulate Free Register Bit, it is possible to generate a table, indexed by the particular thread’s PC that contains flags indicating whether either of an instruction’s registers operands were last used. On each simulated instruction, the simulator preformed a lookup in this table to determine whether register de-allocation should occur when the instruction is retired;

2) **Free Register** – instead of specifying last uses in the instruction itself, it uses a separate instruction to specify one or two registers to be freed. A very light modification to the compiler allows it to generate a Free Register instruction, using the NOP code in SMT MIPS immediately after any instruction containing a last register use (if the register is not also redefined by the same instruction). Like a Free Register Bit, it frees registers as soon as possible, but with an additional cost in dynamic instruction overhead;

3) **Free Mask** is an instruction that can free up to 32 registers, and is used to deallocate dead registers over a large sequence of code, such as a basic block or a set of basic blocks. Rather than identifying dead registers the operand specifies, the compiler generates a bit mask. The mask uses the lower 32 bits of the register to indicate which registers must be freed. This mask is generated and loaded into the register using a pair of 16-bit immediate values. Free Mask sacrifices the rpmphness of Free Register’s deallocation for a reduction in instruction overhead. Free Opcode is somewhat motivated by the principal of special locality and by the observation [7] that ten opcodes are responsible for about seventy percent of the dynamic instructions with last use bits set in the typical benchmark (this assumption is not necessarily held in a different type of workload). This strategy indicates that most of the benefit of the Free Register Bit could be obtained by providing special versions of those opcodes. In addition to executing their normal operation, the new instruction also specifies that either the first, second, or both operands are last uses. Free Opcode/Mask augments Free Opcode by generating a Free Mask instruction at the end of each trace. This hybrid scheme addresses register last uses in instruction that are not covered by particular compiler choices of instructions for Free Opcode.
The second and the third strategies can be also mimicked by the kernel, making it produce either Free Register or Free Mask instructions and inserting them into the pipeline for all threads in the current context.

### 3.6.3. SMT MXS Support for Register Renaming

The SMT MXS keeps the following information about each physical register:

1) **reg_status.** Status flags;
2) **reg_ref.** During renaming, if there is a dependency on a previous instruction in the window the reference count will be incremented by one;
3) **reg_nmap.** Map count for register. Both this and the next member are simple reference counters;
4) **reg_nclaims.** Claim count for register.

Each physical register can have a combination of seven states. These states are:

1) **REG_BUSY.** The register is busy; the register is not busy if it is not mapped, and it is not in the instruction window;
2) **REG_IN_WIN.** The register is in the instruction window;
3) **REG_MAPPED.** The register is mapped;
4) **REG_DMAP.** The register is doubly mapped (i.e., it is being mapped and in the instruction window);
5) **REG_ERROR.** If a source register has the REG_ERROR bit set, this indicates that the writer of the register caused a fault, so stall this thread until the problem is dealt with;
6) **REGCLAIMED.** The register belongs to thread;
7) **REG_FREED.** The register is freed.

All SMT MXS support for register renaming can be found in the file sim_ms_fetch.c.

The SMT MXS uses the following functions to implement register renaming:

1) **sim_ms_rename.** Rename logical registers to use real (physical) registers. Register 0 is never remapped, and registers are allocated in pairs. This function returns -1 if rename fails (out of free registers, or special register still in use). If a source register
has the REG_ERROR bit set, this indicates that the writer of the register caused a fault, so stall this thread until the problem is dealt with;

2) AcquireRegMap. The function marks physical register mapped, and track number of times it is mapped;

3) ReleaseRegMap. This function decrements map count for register and mark it unmapped when the count reaches 0;

4) CheckRegFree. This function updates register state. For normal registers, they become free when the number of references drops to zero, and there is no other reason to stay busy (i.e. they aren't still mapped and aren't in the instruction window). In the PRECISE case, when the writer of a register graduates, the prior name for that register is available. Setting the REG_FREED flag indicates this. For special registers, they stay mapped all the time, so just check if they are still in the instruction window. For claimed registers (the other half of double precision registers), unclaim them if writeback has happened, the number of claims is zero, and the number of references is zero. This guarantees that a subsequent claimant can write to the register without interfering with any thread relying on a prior value in that register. The register number for this routine has already been divided by 2.

5) InvalidateMap. This function invalidated current renaming map in order to provide support for the OS CDI instruction. For more details on why this function is needed, please refer to the previous section of this work.

Indeed, the most interesting function of these three is sim_ms_rename. Below, there will be provided more implementation details on this function.

The function begins the execution by checking the instruction members for errors, and the status of the instruction to be suitable for renaming. If any of the checks present a problem, the function returns with error status of -1. Otherwise, a new free register is acquired by making a call to the AcquireRegMap, which increment the reference count of this register and places the register into the REG_BUSY state. In the case when the renaming operation was invoked by the speculative thread (REGCLAIMED state), an additional check is provided in order to verify if the register is being renamed by the instruction that is on a correct execution path of the branch. For more details on problems associated with branch speculative execution please refer to “Branch Prediction Unit of SMT
SimOS" section of this work. For special registers renaming process, the function also serializes all accesses to these special registers. The last item that the function does is it updates the CPU state. If no errors occurred during the execution, the function returns 0 as indication of its success, along with the new register number assigned to the physical register. For more details on updating the CPU state and the s_cpu_state structure please refer to the "Branch Prediction Unit of SMT SimOS" section of this work.

3.7. SMT Issue Queues

In original R10000 architecture any instruction that was decoded in stage 2 of the SMT MIPS pipeline is appended to one of three instruction queues: integer, address and floating-point queue.

The three instruction queues can issue one new instruction per cycle to each of the queues. The queues allow the processor to fetch instructions at its maximum rate, without stalling because of instruction conflicts or dependencies. Each queue uses instruction tags to keep track of the instruction in each execution pipeline stage. These tags set a Done bit in the active list as each instruction is completed.

3.7.1 Integer Queue

The integer queue issues instructions to the two integer arithmetic units: ALU1 and ALU2. The integer queue contains 16 instruction entries. Up to four instructions may be written during each cycle: newly-decoded integer instructions are written into empty entries in no particular order. Instructions remain in this queue only until they have been issued to an ALU. Branch and shift instructions can be issued only to ALU1. Integer multiply and divide instructions can be issued only to ALU2. Other integer instructions can be issued to either ALU. The integer queue controls six dedicated ports to the integer register file: two operand-read ports and a destination write port for each ALU.
3.7.2 Floating-point Queue

The floating-point queue issues instructions to the floating-point multiplier and the floating-point adder. The floating-point queue contains 16 instruction entries. Up to four instructions may be written during each cycle; newly-decoded floating-point instructions are written into empty entries in random order. Instructions remain in this queue only until they have been issued to a floating-point execution unit. The floating-point queue controls six dedicated ports to the floating-point register file: two operand-read ports and a destination port for each execution unit. The floating-point queue uses the multiplier’s issue port to issue instructions to the square-root and divide units. These instructions also share the multiplier’s register ports. The floating-point queue contains simple sequencing logic for multiple-pass instructions such as Multiply-Add. These instructions require one pass through the multiplier, then one pass through the adder.

3.7.3 Address Queue

The address queue issues instructions to the load/store unit. The address queue contains 16 instruction entries. Unlike the other two queues, the address queue is organized as a circular FIFO buffer. A newly decoded load/store instruction is written into the next available sequential empty entry; up to four instructions may be written during each cycle. The FIFO order maintains the program’s original instruction sequence so that memory address dependencies may be easily computed. Instructions remain in this queue until they have graduated; they cannot be deleted immediately after being issued, since the load/store unit may not be able to compete the operation immediately.

The address queue contains more complex control logic that the other queues. An issued instruction may fail to complete because of a memory dependency, a cache miss, or a resource conflict; in these cases, the queue must continue to reissue the instruction until it is completed.

The address queue has three issue ports: it issues each instruction once to the address calculation unit. This unit uses a 2-stage pipeline to compute the instruction’s memory address and to translate it in the TLB. Addresses are stored in the address stack and in the
queue's dependency logic. This port controls two dedicated read ports to the integer register file. If the cache is available, it is accessed at the same time as the TLB. A tag check can be performed even if the data array is busy.

The address queue can re-issue accesses to the data cache. The queue allocates usage of the four sections of the cache, which consist of the tag and data sections of the two cache banks. Load and store instructions begin with a tag check cycle, which checks to see if the desired address is already in cache. If it is not, a refill operation is initiated, and this instruction waits until it has completed. Load instructions also read and align a doubleword value from the data array. This access may be either concurrent to or subsequent to the tag check. If the data is present and no dependencies exist, the instruction is marked done in the queue.

The address queue can issue store instructions to the data cache. A store instruction may not modify the data cache until it graduates. Only one store can graduate per cycle, but it may be anywhere within the four oldest instructions, if all previous instructions are already completed. The access and store ports share four register file ports (integer read and write, floating-point read and write). These shared ports are also used for Jump and Link and Jump Register instructions, and for move instructions between the integer and register files.

### 3.8. Branch Prediction Policies

As indicated by [8], recent CPU architectures implemented using the increasing degrees of instruction level parallelism. The point here is that with any chosen implementation (including superpipeline, Superscalar and speculative execution techniques) branch instructions are increasingly important in determining overall machine performance. Moreover, some of compiler-assisted techniques for minimizing branch cost in early RISC designs have become less appropriate or even obsolete. The branch prediction policies can be classified as **static** and **dynamic**. The following static branch prediction policies are the most popular:
1) **Always not-taking** – every time a branch instruction is issued, it is assumed not to be taken and execution continues normally with the next instruction. If the branch is taken, the pipeline is flushed to remove the incorrectly fetched instructions and execution resumes with the correct target address;

2) **Always taken** – this is the same scheme as the always not-taken scheme, but with the assumption that the branch instruction is always taken and execution continues with the target address;

3) **FTBN (Forward taken, backward not-taken)** – if target address is forward in direction, then the branch is taken. Otherwise, it is not taken;

4) **FNBT (Forward not-taken, backward taken)** – if target address is forward in direction, then the branch is not taken. Otherwise, it is taken.

Those strategies are well exploited and do not depend on the nature of the workload. Therefore, the focus will be set on the dynamic branch prediction policies. There are two questions that have to be answered in order to resolve the branch performance problem: how to predict the branch direction and how to minimize the execution delay of a target instruction for taken branches. One way to quickly provide the target instructions is to use a Branch Target Buffer, which is a special instruction cache designed to store the target instructions. The approach utilizes the principal of address locality, which states that a particular set of addresses, once used, is likely to be used again in the nearest future by the same context. First, let’s consider an answer to the first question. Then this work will focus on the second problem and will provide few solutions to decrease delays associated with the issue of target instructions in the taken branches.

The most well known technique for determining the direction of the branch, namely the **bmodal** branch prediction, makes a prediction based on the direction the branch went the last few times (usually up to four) it was executed. Experiments have proven that utilizing significantly more branch history can make a higher number of accurate predictions.

The **history table** takes advantage of addresses time locality. Since the histories are independent, this method was called the local branch prediction policy.
Another technique uses the combined history of all recent branches in making a prediction (this approach utilizes the principal of special locality, in which the closely located addresses are likely to be used identically by the same context. A typical example of this principal is the loop statement). Some CE sources call this technique the **global branch prediction** policy.

Each of these different branch prediction methodologies has distinct advantages. The bimodal technique works well when each branch is strongly biased in a particular direction (table structures). The local prediction policy works well for branches with simple repetitive patterns (while and for loops). The global prediction scheme works particularly well when the direction taken by sequentially executed branches is highly correlated (stack operations).

In addition, [8] introduced a new **hybrid technique** that allows the distinct advantages of different branch predictors to be combined. The technique uses multiple branch predictors and selects the one, which is performing best for each branch. This approach is shown to provide more accurate predictions than any one predictor alone. However, the method described in [8] that allows increasing the utility of branch history by hashing it together with the branch address will be considered.

Provided that there are no problems with the implementation of a branch, the branch is either taken or not taken. The bimodal branch prediction takes advantage of this bimodal distribution of branch behavior and attempts to distinguish the usually taken branches from the usually not-taken ones.

In the SMT context, the simplest way to do it is to implement the number of tables of counters indexed by the low order address bits in the program counters or alternatively use a single table of counters that are shared not only by the branches, but also by the contexts.

Each counter is two bits long. For each taken branch, the appropriate counter is decremented. In addition, the counter can saturate. The typical counter range is from 0 to 3. The most significant bit determines the prediction. Repeatedly taken branches will be predicted to be taken, and repeatedly not-taken branches will be predicted to be not taken. All counter values between 0 and 3 tolerate a branch going an unusual direction one time.
For large counter tables, each branch will map to a separate counter, whereas for smaller tables, multiple branches may share the same counter. The second approach suffers from decreased accuracy of prediction. As an alternative solution, it is possible to store a process id and the tag with each counter and use a set-associative lookup to match counters with branches. The [8] showed that for a fixed number of counters a set-associative table has a better performance. However, once the size of tags has been determined, a simple array of counters often has better performance for a given predictor size.

To improve the accuracy of the bimodal policy, consider the typical compiler loop unrolling action. In a simple for loop, the instruction that does the loop test is placed at the end of the body. The corresponding bimodal pattern will look like \((1110)^n\), where the last two bits 1 and 0 represent the taken and not-taken cases respectively, and \(n\) is the number of time the loop is executed. If the direction this branch had gone on the previous three executions had been known, then it would be possible to predict the next branch direction.

Consider the predictor that uses two tables. The first table records the history of recent branches. This work uses the hash of simple arrays indexed by the low-order bits of the branch address. The hash key indicates the context, in which the particular set of branches is executed. The lookup is set-associative. Like the bimodal scheme, this approach avoids the need to store tags but does suffer from degraded performance when multiple branches map to the same table entry, especially with smaller table sizes.

Each history table entry records the direction taken by the most recent \(n\) branches whose addresses map to this entry per context, where \(n\) is the length of the entry in bits. The second table is an array of 2-bit counters identical to those used for bimodal branch prediction. However, here they are indexed by the branch history stored in the first table.

The advantage of this approach is that the only first table is changed in regard to the SMT. The second table does not require any additional changes, since there is 1:1 relation between each history table entry in the first table and each prediction counter in the second table. With this approach the history used is local to the current branch. Therefore, it is reasonable to call this scheme the local prediction policy.

The local predictor, however, might be affected by two kinds of contention. First, each branch history may reflect a mix of histories of all the branches that map to each history
entry. Second, since there is only one counter array for all branches for all the contexts, there may be conflict between patterns. As the number of history bits increases, the second condition may be avoided, but the table can no longer dynamically adjust to the more recent patterns, as it used to be able in the 3-bit history case. In the local branch prediction scheme, the only patterns considered are those of the current branch.

Another approach is to take advantage of other recent branches to make a prediction. A typical implementation of the strategy includes the set of shift registers GR that record the direction taken by the most recent n condition branches generated by a context. There are two types of patterns, of which the global branch prediction is able to take advantage. First, the prediction of a particular branch may strongly depend on the other branch predictions made earlier (for example, in multiple inclusions). Second, the global branch predictor can be effective by duplicating the behavior of local branch predictor. This can occur when the global history includes all the local history needed to make an accurate prediction.

The scheme is effective when working with the current branch. This suggests that the more efficient prediction might be made using both the branch address and the global history. In this approach the counter table is indexed with a concatenation of global history and branch address bits. The performance of this scheme depends on a tradeoff between using more history bits or more address bits. The [8] refers to this strategy as to **global predictor with index selection** policy. It is clear that the previous policy weakly identifies the current branch. If there is enough address bits to identify the branch, the frequent global history combinations should be rather sparse. Therefore, [8] suggests hashing the branch address and the global history together by OR-ing their bits. Since both address and global history bits are in use, this strategy leads to lower misprediction rate. If fewer global history bits were chosen instead of branch address bits, an exclusive OR on the global history bits and the higher order address bits would have to be done, since the higher order address bits will be sparser than the lower order bits.

The **McFarling's branch prediction** scheme has different advantages. The scheme is flexible combining different advantages to achieve the better performance depending on the nature of the workload. The policy combines two predictors: p0 and p1 that could be one of the predictors discussed previously or any kind of branch prediction method. In addition,
the combined predictor contains an additional counter array, which serves to select the best predictor to use. As before, the scheme uses 2-bit ranged counters. Each counter keeps track of which predictor is more accurate for the branches that share that counter. Specifically, [8] uses the notation c (p0c, p1c) to denote whether predictors p0 and p1 are correct respectively, the counter is incremented or decremented by the subtraction p0c-p1c. Before proceeding with developing of new SMT branch prediction unit, provided is a short discussion on minimization of taken branch target delays.

A Simple Branch Target Buffer is usually used to remember as many as possible of the taken branches that are encountered in the dynamic instruction stream. To mask the penalty of flushing the instruction fetch unit, the buffer stores the first k instructions of a taken branch’s target path. For this reason, any branch instruction not in the target buffer is predicted to be not-taken. In case when a branch instruction is predicted taken, but when executed it does not branch to a new location; the corresponding entry in the buffer is deleted. Therefore, the buffer acts like a cache that uses the branch instruction’s location in memory as its associative tag. When it is full, a replacement policy is used to select an entry to replace.

Similarly to the Simple Branch Target Buffer, a Counter-based Branch Target Buffer is also a type of cache. It remembers as many the branch instructions encountered in the dynamic instruction stream as possible. As with the simple buffer, the counter-based buffer also stores the first k instructions of the target branch to mask the instruction fetch penalty. For each new entry in the table, the n-bit counter C is initially set to a threshold T if the branch was taken, or T - 1 if it was not taken. Subsequently if the branch is taken, the counter is incremented, or else it is decremented. The counter saturates at 0 and at 2^n - 1. A branch is predicted taken when C ≥ T, else the branch is predicted not-taken. Any branch instruction not already in the buffer is predicted not-taken.

Both Simple and Counter-based buffers are accessed using the address from the select stage of the instruction fetch unit for every instruction retrieved from memory for every context. This access occurs in parallel with the actual memory access performed in the instruction fetch unit (see the “Fetching Unit” section for more details). If the location causes a buffer hit (the “valid” bit is set and the address matches the tag), it is then known
that the instruction is a branch. If the Simple or Counter-based buffer predicts the branch as taken (hit is usually predicted as a taken branch), the first $k$ instructions following the target are sequentially supplied to the instruction decode unit.

The third approach to branch prediction, the **Forward Semantic**, uses an optimizing, profiling compiler to predict the direction of all branches in a program. The program is first compiled into an executable intermediate form with probes inserted at the entry of each basic block. The program is then run once or several times for a representative input suit. All branch instructions are converted to the format that allows an additional "likely-taken" bit. During the recompilation, predictions are make for each branch and stored by setting or clearing this bit in the instruction format of each branch instruction. Based on the profiling information, groups of basic blocks that are virtually always executed together are then bundled into larger blocks called traces. The result is that all conditional branches that are predicted taken are placed at the end of the traces. The compiler reserves $k + 1$ locations for each branch that is predicted taken. Those are called forward slots. The appropriate number of instructions from the target path is then copied into these slots. When the instruction is determined to be a branch instruction at the end of the instruction decode stage, the instructions in the forward slots will mask the penalty of incorrectly fetching the $k + 1$ instructions following the branch. To fill the forward slots, the traces are sorted by execution weight that was derived from McFarling’s scheme, but also was extended to support SMT.

All branch prediction support functions can be found in smtbranch files. The [9] proposes the implementation described below. However, the SMT implementation is completely different, as well as some configuration details. In [9] implementation, branch_pred.h is the header file that defines all the interfaces, branch_parm.h defines the default parameters for all the above schemes of branch prediction (the corresponding SMT MXS simulator file should probably be moved closer to global configuration file of the SimOS), which also specifies, along with other parameters, the option to turn off the prediction unit and the branch predictor scheme, branch_pred.c – implements all the functions needed for prediction.

The unit configuration includes: module type – specifies SMT or usual dynamic (and only dynamic) prediction method; counter table size – specifies the table size for 2-bit
Counter model; Shift register size – specifies number of bits in shift register for any two level model, L1 table size – specifies the number of entries in first level; L2 – specifies the number of entries in second level; BTB sets – specifies the number of sets for BTB; BTB associativity – specifies degree of associativity of BTB. The unit implements the following functions:

1) pred branch_init(type) – the function reads parameters, creates and initializes a branch predictor instance. Input: type – branch predictor schemes. Output: pred – branch predictor instance;

2) long branch_pred(pred, addr) – the function predicts a branch target address, given a branch instruction address. Input: pred – branch predictor instance, addr – branch instruction address;

3) bool branch_update(pred, addr, real_target, pred_taken, really_taken, addr_pred_correct – the function updates branch predictor after real branch target is resolved. Input: pred – branch predictor instance, addr - branch instruction address, real_target – resolved real branch target, pred_taken – true, if address was predicated taken, otherwise false, really_taken – true, if branch was really taken, otherwise false, addr_pred_correct – true, if the address prediction was correct, otherwise false. Output – true if successfully updated, otherwise false;

4) void branch_monitor(pred, val, id, psize) – the function monitors the branch prediction units. Input: pred – a branch predictor instance, void * val – returned structure that contains monitored values, long * id – the array of identifiers. Each of these identifiers corresponds to a single value from val. The structure and the array are passed to the global monitor that formats the statistics for each particular id, psize – by reference, the size of the array or an array of the dimensions of the array (the lower bound is always considered to be 0);

5) void branch_delete(ppred) – the function disposes a branch predictor instance and frees spaces allocated. Input: ppred – by reference, pointer to a branch predictor instance. If resource are successfully deallocated the (*ppred) becomes 0, otherwise it remains non-zero.
3.9. Branch Prediction Unit of SMT SimOS

All the functionality of the branch prediction unit is implemented in file sim_ms_branch.c, function ms_branch. All branch instructions are handled by spawning a number of threads. Each thread follows one predicted path. The maximum number of active threads per branch is controlled by MAX_ACT_THREADS parameter. If this number is equal to one, no threads are available. In this case no speculative execution occurs. If a branch is conditional, the branch prediction logic is invoked. The newly spawned thread follows the predicted path, while the original thread holds the state prior to the branch. If multiple active threads are allowed, then both threads execute simultaneously. When a particular branch needs to use the prediction logic, it does not create a new thread, but takes it from the pool of available threads. The maximal number of available threads is controlled by THREAD_WIDTH parameter. This number is usually calculated by multiplying the maximal number of threads in a single SMT context by the number of possible prediction paths (this number equals to 2). Assuming 8 threads in context (for more details see “Fetching Unit”), the THREAD_WIDTH parameter is set to 16.

In order to understand the function logic, it is necessary to discuss the following structures, which are defined in sim_ms.h:

1) s_cpu_state structure that defines the state of MXS – compatible simulated CPU models,
2) THREAD (or s_thread) structure that represents the thread on a branch prediction path,
3) INST (or s_inst) structure, which is associated with a single CPU instruction,
4) BrTREE structure that represents an entry into a Global Branch History table.

The s_cpu_state structure contains the following members (for simplicity, only those members that are related to the function discussed are specified):

1) bp_bits. This member is an array of the saturated counters, as it was described in the previous section. This array is always of BP_TABLE_SIZE size. Since the array represents both Global and Local Branch History counter arrays, it should be large enough to provide the simulator with accurate measurements. The BP_TABLE_SIZE constant is set to 1024 in sim_params.h file.
2) **bp_total_accesses.** This member represents the total number of accesses to a particular saturated counter in the Global Branch History table. Naturally, the member is an array of BP_TABLE_SIZE size. As it was described in the previous section, every saturated counter is indexed by the low n bits of an accessed virtual address.

3) **bp_hits.** This is also an array of BP_TABLE_SIZE size, each member of which represents a total number of times the counter indexed by a particular set of virtual addresses correctly predicted the direction of a branch.

4) **bp_mfm.** This member also holds BP_TABLE_SIZE elements, each of which represents the McFarling's metric. For more details on McFarling's metric see the previous section. The simulator updates the array by calling McFarlingMetric function. The function accepts the s_cpu_state structure as its input parameter. The function simply subtracts the number of hits for each set of virtual addresses from the number of total accesses to the set.

5) **old_WRITE_w.** This member represents the array of BP_TABLE_SIZE of WRITE weights. To get more information about this member, the reader should refer to the “WRITE Branch Prediction Policy” section of this work. Briefly, the WRITE (Weighted RIT Extremum) branch prediction policy allows accurate calculation of the direction of a branch. Unlike all standard algorithms, the WRITE policy can automatically adjust itself to provide the most accurate branch prediction at any given time during simulation.

6) **bp_targets.** This member corresponds to an array of BP_TABLE_SIZE size elements, each of which represents a Branch Target Buffer for a particular counter.

7) **branch_stack.** This member represents the stack of the BP_RETURN_STACK size. It is needed to execute the Return From Subroutine instruction.

Among other members, the THREAD structure contains the following members related to the function in focus:

1) **pc,** a program counter per thread,

2) **thread_st.** This member takes its values from enumerated set of states of a thread. This member is updated, among a few other structures, by AddToStat and IncStat macros.
3) stall_fetch. When the ms_fetch function is discussed, there is also a reference to this member. The member is actually the flag, which represents whether or not the fetch unit has been stalled. Among several reasons, the branch misprediction can cause the fetch unit of a particular thread to stall.

4) branch_dly. The member is the flag that indicates that a thread is executing instruction within a branch delay slot.

5) branch_likely, the same as the previous one, but with regard to branch likely condition.

6) returnpc. This member is associated with the new value of the pc counter on return from a subroutine call.

7) branch_sp. The member is assigned the value of a particular branch stack pointer.

8) old_prediction – before the branch stack gets its new value, an old value of the branch stack is assigned to this member.

9) branch_inum – each branch instruction within a thread is indexed with some integer number, which is saved in this member.

10) branch_node, this member allows to index a particular BrTREE node within a thread.

11) branch_pc – this member gets the calculated value of a branch target instruction. Usually the first k instructions, starting from this address, are saved in Branch History buffer.

12) branch_likely_pc – this member has the same functionality as the one previously described, but with regard to branch likely instruction execution.

The INST structure includes an operating code and a few special entries that allow fast processing of an instruction.

The BrTREE structure represents an entry to the global binary tree. The tree itself represents the Global History Table with the number of recorded entries equal to two. An important node: in order to create a Global History Table that stores more than two entries at a time, the BrTREE structure must be reorganized to form N-ary tree, where N represents the number of historical entries for every branch that is being predicted. The structure contains the following members:

1) thread. This member represents the thread number of the node.

2) thread_st, the status of the thread.
3) lchild, rchild – indices of the left and right children.
4) resolution. This flag indicates how to change the Global History Table depending on how the last branch instruction has been resolved.
5) indirect – the flag that indicates whether a branch instruction is an indirect branch instruction. This flag is set or reset by a call to an is_immediate macro.
6) jret – this flag indicates if a branch instruction is the return from subroutine call.
7) call – an indicator of whether a branch instruction is a subroutine call.
8) uncond – the member represents a flag that is set or reset by a call to an is_immediate macro. The flag indicates if a branch instruction is an immediate instruction.
9) restore – the flag that indicates if the stack needs to be restored upon the completion of execution of a branch instruction.

Having described the structures, it is feasible now to proceed with further discussion on the ms_branch function.

When the next branch instruction is fetched, and its operation code is determined, the ms_branch function is invoked. The function begins by acquiring the index of the executed branch instruction in the thread instruction window.

The next step is to retrieve the actual instruction by getting its address from the iwin array by the instruction index branch_inum. This operation is guaranteed to succeed, since the branch instruction is present in both the instruction window and the Branch Prediction binary tree. The actual address of the tree node is retrieved from the branch_tree array of the s_cpu_state structure by its index branch_node. The function then memorizes the thread structure associated with the branch (the thread member of the tree node). In order to continue the processing, a number of additional checks must be made. The simulator provides a number of useful macros to verify if a particular branch instruction is indirect (is_indirect_branch macro), a return from subroutine call instruction (is_call macro), or is an immediate unconditional branch instruction (immediate_unconditional macro). The above listed macros may have also been implemented as functions, but it would have added an unnecessary overhead to the total execution time of the simulator.

Once the type of the branch instruction has been identified, the function is ready to act based on this information. In the case when the branch instruction is the immediate, but not the subroutine call, instruction, the function simply executes it. The reason for that is that
the instructions of this type do not require any branch predictions. Therefore, the function marks the instruction as available and puts it in the end of the issue queue (ms_pri_dequeue function call). At this point, the state of the CPU is switched to a “Branch Execution”, the thread process counter is assigned the immediate branch instruction address offset, and the target instruction is retrieved in the way similar to the one for the branch instruction that was described above.

There is nothing else that needs to be done with the instruction, after it has been placed in the issue queue. However, the simulator work frame should not be forgotten at this point. The specifics of any simulator is that the simulator, unlike any real CPU, does not directly support neither BREAK instruction, nor any instruction latency measurements. Due to this fact, another function must be defined, namely, ms_break, which would hold the further execution and output the break mode information on the screen.

The ms_break function is invoked only if the BREAKPOINT constant is defined during the simulator’s compile time. Similarly, the set of latencies is defined for all instructions supported by a particular simulator in sim_params.h file and take those into consideration, if the TAKEN_LATENCY compile time constant is defined. How does the simulator implement the latency mechanism? It defines another structure called “work list”. This structure contains a hash of instructions scheduled for issue and a set of associated counters. As an instruction is inserted into the hash (Add_to_worklist function call), the associated latency is retrieved for the instruction from the list of the instruction latencies, and the associated timer is started. Once, the value of the timer reaches 0, the instruction is removed from the hash and is enqueue in the issue queue. Both BREAKPOINT and TAKEN_LATENCY execution branches are completed by the finish_branch function call, which completes the processing of the branch instruction.

A more difficult case arises when the branch instruction requires invocation of the branch prediction mechanism. In this case the ms_branch function has to acquire another thread for speculative execution of the branch instruction. The number of threads is limited, as it was described above, due to the operating system limitations. Therefore, is the ms_branch function makes a request for a new thread, and the pool of available threads is empty, the simulator stalls the execution of the branch instruction and swaps the context out
of pipeline. For more details on how the simulator does it, refer to the "SMT CPU Model" section of this work. If the ms_branch function successfully acquires an ownership of the new speculative thread, it initializes all associated members of the thread with required values, including the process of register renaming. For more details on register renaming, refer to the "Register Renaming and Special Techniques" section of this work.

Once the new thread has been initialized, the lchild and rchild members of the branch tree node structure are initialized with the indices of the historical entries from the Global History table. At this point, if the branch is unconditional, only the branch instruction is handled. Otherwise, if the branch is conditional (a call to the is_conditional macro returns TRUE), branch prediction is done. The first step in the processing requires the update to the state of the CPU to "Processing Conditional Branch". Then the values of the saturated counters for all of the three branch tree nodes are retrieved. The call to PredictTaken macro returns the predicted direction of the branch instruction. If the call returns TRUE, the branch is predicted as taken. Otherwise, the branch is predicted as not taken. An important thing to understand here is that though the PredictTaken macro returns a binary value, it also updates the pbits value. This value is the combined value of all of the three saturated counters. This value is non-binary and can take any values between 0 and BP_TAKEN + BP_BOTH, where the BP_BOTH constant corresponds to the "not-taken" threshold, and the BP_TAKEN constant corresponds to the "taken" threshold of a branch prediction. Therefore, if the value of the pbits is less than the BP_BOTH constant, the branch is considered to be not-taken. On the other hand, if the value of the pbits is equal or greater than the sum of the BP_TAKEN and the BP_BOTH constants, the branch is considered to be taken. The described above cases has a common name: "strong prediction". In both cases, one of the two threads must be shut down and returned to the pool of the free threads. The idea here is that the two threads complement each other: if the first thread is responsible for execution of non-taken path, the second thread is responsible for execution of taken path, or vice versa. Therefore, it is necessary to kill one or the other thread, when dealing with a strong prediction. Otherwise, both threads remain active till the correct direction of the branch is determined. The function kills an appropriate thread by making a call to an inactivate_thread function.

If this point were reached in the ms_branch function, then the system would end up with two active threads executing the same algorithm simultaneously. They both calculate

70
the new instruction addresses. If any of the new addresses corresponds to the register, a new target address is also calculated. When the direction of the branch is determined, the complimentary thread is killed, and the remaining thread exits normally by issuing the first instruction on the correct calculated execution path.

At this point, all that is left is to get an understanding of the process of updating of a particular saturated counter. This is done in sim_ms_inst.c file, where the following two functions are located: opBRANCH and opCALL. The file is generated automatically from sim_ms_inst.m4 file. The sim_ms_inst.m4 file is the GNU m4 preprocessor macrofile. For more details on GNU m4 utility please refer to the GNU documentation. The sim_ms_inst.m4 file defines the CurrentPredictionPolicy constant. This constant controls the current branch prediction policy for the SMT MXS simulator. This constant can be assigned any value from enumerated set of branch prediction policies BP_POLICY. For simplicity, the variable is set to an ALWAYS_TAKEN value from the BP_POLICY set. Since both macros use similar implementation of the process of updating of the saturated counter, let us consider only the first macro, namely opBRANCH.

If the CurrentPredictionPolicy constant is set to ALWAYS_TAKEN, the macro returns the value of BP_TAKEN. If the CurrentPredictionPolicy is set to ALWAYS_NOT_TAKEN, the macro returns the complementary value of ~BP_TAKEN. Otherwise, the decision is made in favor of either left or right branches, and the returned parameter get its values from the interval described above.

The macro also utilizes the McFarlingMetric and WRITE_ functions. The first function was already discussed, which keeps track of McFarling’s matrices of branches. The second function will be discussed in great detail in the following section.
3.10. WRITE Branch Prediction Policy

All branch prediction policies discussed so far lack one important property: flexibility. Even McFarling's prediction strategy can only report comparative accuracy of prediction at any time during the execution.

Imagine the branch prediction policy that adjusts its predictions based on accuracy of all prediction policies supported by the simulator. Every time an attempt to predict the direction of a branch is made, the simulator acquires predictions from all policies, picks up the ones that performed the best during the last round of prediction, calculates the new value of the saturated counter based on those predictions, and waits till the correct direction of the branch is determined. Once the correct direction of the branch has been calculated, the simulator then encourages the policies that have been proven to predict correctly by adding to their associated weights, and discourages those policies that happened to predict incorrectly. As the result, the simulator always keeps close to the accuracy extremum of branch predictions, utilizing the special and time locality principals in approximately the best way.

The WRITE function returns the new values of the weights of the supported branch prediction policies. The actual calculation is done in a way analogous to the neural networks weight re-calculation. As it was described above, the word WRITE stands for "Weighted RIT Extremum", and the policy itself forms a good foundation for other scientific research.

3.11. Exception Handling in SMT MXS

All R10000 integer exceptions may be divided into four categories [10]:

1) address error exceptions, which occur when a data item that is referenced is not properly aligned or invalid for the executing process;
2) overflow exceptions, which occur when arithmetic operations compute signed values and there is not enough precision in the destination to hold the result;
3) bus exceptions that happen when the address is invalid for the executing process;
4) divide-by-zero exceptions.
Those, with some additions, are applicable to floating point exceptions. The processor also generates an interrupt (exception 0).

The priority of exceptions the reader can find in [5]. The priorities do not change, as R10000 is switched to SMT CPU architecture. Also, the processing of the first three exceptions (hard, soft resets and the non-maskable interrupt) remains unchanged and has already been implemented in MXS (should all be redirected to Cold Reset: this should be checked). These exceptions are stored in uncached and unmapped addresses. The rest of the exceptions require massive changes due to the fact that in the SMT architecture any exception handling should attempt to do its work without context switch. Those are a combination of a vector offset and a base address. The InstallErrorHandler function has been implemented that follows the standard procedure:

1) BEV=0 bit check in the Status register is done to assure the normal mode of operation;
2) The addresses of all the exception handlers are taken from the exceptions_addrs structure.

The exception_addrs structures has the following three members:

1) exc_cpu_addr – the address from the Table 17-1 [16], which indicates the jump CPU address;
2) enum exc_id – parameter that identifies the exception type to the kernel. The following sections provide both the exc_id and the description for each exception handled by the SMT MIPS architecture;
3) exc_handler – the kernel handler jump address.
3.11.1. Cold Reset Exception

The interrupt vector for this exception is located at 0xBFC00000 in 32-bit mode or in 0xFFFFFFFFF0000000 in 64-bit mode. The Cold Reset vector resides in unmapped and uncached CPU address space, so the hardware need not initialize the TLB or the cache to process this exception. It also means the processor can fetch and execute instructions while the caches and virtual memory are in an undefined state. The contents of all registers in the CPU are undefined when this exception occurs, except for the following register fields: in the Status register, SR and TS are cleared to 0, and ERL and BEV are set to 1. All other bits are undefined.

Config register is initialized with the boot mode bits read from the serial input; the Random register is initialized to the value of its upper bound; the Wired register is initialized to 0; the EW bit in the CacheErr register is cleared; the ErrorEPC register gets the PCs of the first context; the FrameMask register is set to 0; branch prediction bits are set to 0; Performance Counter register Event field is set to 0; all pending cache errors, delayed watch exceptions, and external interrupts are cleared. The exception is serviced by do_cold_reset kernel function. This function contains two parts:

1) hardware dependent part that initializes all processor registers, coprocessor registers, caches, and the memory system;
2) topsy bootstrapping part. For more information on Topsy bootstrapping process see the “SMT Topsy” sections of this work.

For simplicity, the do_cold_reset function omits all CPU diagnostic tests, including initialization of the CPU performance counters. The function do_soft_reset() reads the processor state and then proceeds with the standard Cold Reset sequence:

1) preserves all contents of all registers except for ErrorEPC registers, ERL bit of the Status register, which is set to 1, SR bit of the Status register, which is set to 1 on Soft Reset or an NMI 0 for a Cold Reset, BEV bit of the Status register, which is set to 1, TS bit of the Status register, which is set to 0, kernel PC (PC register) is set to the reset vector 0xFFFFFFFFF0000000;
2) clears any pending Cache Error exceptions. NMI exceptions are served by the same function, except that the function do_cold_reset has to set the SR bit of the Status register to 1. do_topsy_bootstrap()

3.11.2. Soft Reset Exception

When this exception occurs, the SR bit of the Status register is set, distinguishing this exception from a Cold Reset exception. When a Soft Reset is detected, the processor initializes minimum processor state. This allows the processor to fetch and execute the instructions to the exception handler, which in turn dumps the current architectural state to external logic. Hardware state that loses architectural state is not initialized unless it is necessary to execute instructions from unmapped uncached space that reads the registers, TLB, and cache contents. The Soft Reset can begin on an arbitrary cycle boundary and can abort multi cycle operations in progress, so it may alter the machine state.

3.11.3. Address Error Exception

This exception is invoked if one of the following occurred:
1) reference to an illegal address space;
2) reference the supervisor address space from User mode;
3) reference the kernel address space from User or Supervisor mode;
4) load or store a doubleword that is not aligned on a doubleword boundary;
5) load, fetch, or store a word that is not aligned on a word boundary;
6) same as previous, but for halfword.

The exception is not maskable. The common exception vector (see [5]) is used for this exception. The AddEL or AdES code in the Cause register is set, indicating whether the instruction caused the exception with an instruction reference, load operation, or store operation shown by the EPC register and BD bit in the Cause register. When this exception occurs, the BadVAddr register retains the virtual address that was not properly aligned or that referenced protected address space. The contents of the VPN field of the Context, XContext,
EntryHi and EntryLo registers are undefined. The EPC register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot. If it is in a branch delay slot, the EPC register contains the address of the preceding branch instruction and the BD bit of the Cause register is set as indication. The exception is serviced by do_addr_error_exception() function in Topsy. For more information on this exception handling see “Exception Handling” subsection in the “SMT Topsy” section of this work. One important thing to understand about this exception is that the associated Topsy SIGSEGV signal for SMT is different from the standard UNIX SIGSEGV signal, and probably should have been assigned a different name.

3.11.4. TLB Exception

This exception is the result of one of the following events:

1) TLB Refill occurs when there is no TLB entry that matches an attempted reference to a mapped address space;

2) TLB Invalid occurs when a virtual address reference matches a TLB entry that is marked invalid;

3) TLB Modified occurs when a store operation virtual address reference to memory matches a TLB entry, which is marked valid bit, is not dirty (not writable).

The exception can be handled from either TLB (32-bit mode) or XTLB (64-bit) exception vectors. The TLB refill vector selection is based on the address space of the address (user, supervisor, or kernel) that caused the TLB miss, and the value of the corresponding extended addressing bit in the Status register (see the table for the associated values of the UX, SX, or KX bits). The current operating mode of the processor is not important except that it plays a part in specifying in which address space an address resides. The Context and XContext registers are entirely separate page-table-pointer registers that point to and refill from two separate page tables, however these two registers share BadVPN2 fields (for more details see the TLB format that remains the same for both R10000 and SMT MIPS architectures).

TLB refill exception occurs when there is no TLB entry to match a reference to a mapped address. The do_tlb_refill() function determines which processor space has caused
an exception. An address is in user space if it is in useg, suseg, kuseg, xuseg, or xkuset. An address is in supervisor space if it is in sseg, ksseg, xsseg or kssegg, and an address is in kernel space if it is in either kseg3 or xkseg. Kseg0, kseg1, and kernel physical spaces (xkphys) are kernel spaces but are not mapped. The function checks the EXL bit for the value of 0, sets the TLBL or TLBS code in the ExcCode field of the Cause register. The combination of EPC register and the BD bit in the Cause register shows whether the instruction caused the miss by illegal reference, load operation, or store operation. The BadVAddr, Context, XContext and EntryHi registers hold the virtual address that failed address translation. The EntryHi register also contains the ASID from which the translation fault occurred. The Random register normally contains a valid location in which to place the replacement TLB entry. The contents of the EntryLo register are undefined. The EPC register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot, in which case the EPC register contains the address of the preceding branch instruction and BC bit of the Cause register is set. To service this exception, the contents of the Context or XContext register are used as a virtual address to fetch memory locations containing the physical page frame and access control bits for a pair of TLB entries. The two entries are placed into the EntryLo0/EntryLo1 register; the EntryHi and EntryLo registers are written into the TLB. It is possible that the virtual address used to obtain the physical address and access control information is on a page that is not resident in the TLB. This condition is processed by allowing a TLB refill exception in the TLB refill handler. This second exception goes to the common exception vector because the EXL bit of the Status register is set.

TLB invalid exception – this exception occurs when a virtual address matches an invalid TLB entry (that is the entry, which TLB valid bit was cleared). The function do_invalid_tlb() checks the TLBL or TLBS code in the ExcCode field of the Cause register. The value of 1 indicates if the instruction cause of the exception. Then the usual TLB refill exception processing takes place. Having completed servicing the exception, the function locates the invalid TLB entry by generating the TLBP instruction and replaces it with the valid entry, which Valid bit is set.

The TLB modified exception occurs when a store operation virtual address reference to memory matches a TLB entry that is marked valid but is not dirty and therefore is not
writable. The do_tlb_modified() function checks the Mod code in the Cause register. Unlike any other exception, the TLB modified exception is most entirely handled by the kernel. For more information on this exception refer to the “Topsy” section of this work.

3.11.5. System Call Exception

The system call exception occurs during an attempt to execute the SYSCALL instruction. The control is switched to the kernel, which sets the Sys code in the Cause register and the EPC register contains the address of the SYSCALL instruction unless it is in a branch delay slot, in which case the EPC register contains the address of the preceding branch instruction. If the SYSCALL instruction is in a branch delay slot, the BD bit of the Status register is set; otherwise this bit is cleared. After servicing the exception, the EPC register must be altered so that the SYSCALL instruction does not re-execute; this is accomplished by adding a value of the EPC register before returning. If the SYSCALL instruction is in a branch delay slot, a more complicated algorithm (for more details see the discussion on bottom-half interrupts and wait queues in the “SMT Topsy” section of this work) may be required.
3.12. SMT MXS Coprocessor 0

This section introduces the SMT MXS coprocessor 0 and its instructions.

Table 6. SMT Coprocessor 0 instructions

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMFCz</td>
<td>Doubleword Move From Coprocessor z</td>
</tr>
<tr>
<td>DMTCz</td>
<td>Doubleword Move To Coprocessor z</td>
</tr>
<tr>
<td>LDCz</td>
<td>Load Double Coprocessor z</td>
</tr>
<tr>
<td>SDCz</td>
<td>Store Double Coprocessor z</td>
</tr>
<tr>
<td>MTC0</td>
<td>Move to CP0</td>
</tr>
<tr>
<td>MFC0</td>
<td>Move from CP0</td>
</tr>
<tr>
<td>TLBR</td>
<td>Read Indexed TLB Entry</td>
</tr>
<tr>
<td>TLBWI</td>
<td>Write Indexed TLB Entry</td>
</tr>
<tr>
<td>TLBWR</td>
<td>Write Random TLB Entry</td>
</tr>
<tr>
<td>TLBP</td>
<td>Probe TLB for matching Entry</td>
</tr>
<tr>
<td>CACHE</td>
<td>Cache Operation</td>
</tr>
<tr>
<td>ERET</td>
<td>Exception Return</td>
</tr>
</tbody>
</table>

The registers that must be changed in the SMT context are of the most interest with regard to this work. The SMT processor contains 8 context registers that are read/write registers containing the pointers to entries in the context page table entry PTE arrays; these arrays are operating system data structures that store virtual-to-physical address translations.

When there is a TLB miss, the CPU loads the TLB with the missing translation from an appropriate PTE array. The lookup is done based on ASID field of the TLB, which is shared by multiple contexts and is protected by the lock (see previous discussion on locks). The operating system uses the Context register to address the current page map, which resides in the kernel-mapped segment, kseg3.

The Context register duplicates some of the information provided in the BadVAddr register, but the information is arranged in a form that is more useful for a software TLB exception handler. The format of a single Context register is provided in [5]. BadVPN2 field is written by hardware on a miss. It contains the virtual page number (VPN) of the most recent virtual address that did not have a valid translation. For a 4-Kbyte page size, this format can directly address the pair-table of 8-byte PTEs. For other page and PTE sizes,
shifting and masking this value produces the appropriate address (for additional details on setting up the page size see the Mask register section in [5]).

The PTEBase field is a read/write field for use by the operating system to use the Context register as a pointer into the current PTE array in memory. All 8 Context registers are initialized in do_topsy_bootstrap() function (see “SMT Topsy” section for more details). The status register does not need to be changed, except for the CU bits that control coprocessor accessibility: CU0, CU1, and CU2 enable coprocessors 0, 1, and 2, respectively. If a coprocessor is unusable, any instruction that accesses it generates an exception.

The following operations are now available on the R10000: 1) coprocessor 0 is always enabled in kernel mode, regardless of the CU0 bit; 2) coprocessor 1 is the floating-point coprocessor. If CU1 is 0 (disabled), all floating-point instructions generate a Coprocessor Unusable exception. In MIPSIV, the COP3 instruction is replaced with COP1X; 3) coprocessor 2 is defined, but does not exist in the R10000, and its instructions always generate exceptions, depending on whether the coprocessor is enabled; 4) coprocessor 3 has been removed from the R10000. The associated instruction (COP3) always causes a Reserved Instruction exception, provided that the coprocessor was previously enabled.

The simple (but not trivial) change to the register is to add the fourth “SMT enable” mode. In order to accomplish this, the way isolate all SMT related architectural details must be found. If SMT is disabled, any attempt to use it will generate a Reserved Instruction exception (of cause, another exception needs to be defined to uniquely identify this exception, but this will immediately require the change to the kernel and the CPU exception vector, as well as a number of other changes).

The 32-bit read/write Cause register describes the cause of the most recent exception. In the SMT architecture the BD bit becomes obsolete due to the more sophisticated branch prediction control and handling (see the discussion on branch prediction strategies). The meaning of the CE and the IP bits remains unchanged. The register is now shared among several contexts. In order to achieve the safe read/write operations on the register, it is protected with the lock.

The Exception Program Counter (EPC) register is a read/write register that contains the address at which processing resumes after and exception has been serviced. For synchronous
exceptions, the EPC register contains either the virtual address of the instruction that was the direct cause of the exception. To ensure the safety, this register is also protected with the lock. Another revision and implementation pair of numbers to the CPU Revision register should be added. Also, it is possible to use the Load Link register for writing actual physical addresses for the associated contexts. For safety, this register also must be protected with the lock.

The Xcontext register is a read/write register that contains a pointer to an entry in the page table entry (PTE) array, and operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the operating system software loads the TLB with the missing translation from the PTE array. The Xcontext register no longer shares the information provided in the BadVAddr register. The Xcontext register is for use with the XTLB refill handler, which loads TLB entries for references to a 64-bit address space, and is included solely for operating system use. The operating system sets the PTE base field in the register. Since there is no explicit context information support provided by this register, it has to be duplicated 8 times to be able to serve up to 8 simultaneously running contexts. The registers are loaded by kernel to address currently running contexts' page maps, which reside in the kernel-mapped segment kseg3. The format of the XContext register remains unchanged in the SMT architecture.

The Diagnostic register is a 64-bit register for processor specific diagnostic functions. Currently, this register helps test the ITLB, branch caches, and the branch prediction scheme. In addition, it provides choices for branch prediction algorithms, to help diagnostic program writing. The 12 bits of the Diagnostic register are read-only and are described below:

1) ITLBM – this field is a 4-bit read-only counter. This field is incremented by one for each ITLB miss, and any overflow is ignored. Its value is undefined during reset, and its value is meaningless when used in an unmapped space;

2) BSIdx – this field defines the entry in the branch stack to be used for the latest conditional branch decoded. Its value is meaningless if the latest branch was an unconditional branch;

3) DBRC – this field disables the use of the branch cache; in the case when the SMT mode is enabled this bit disables the branch predictor;
4) BRCV – this field indicates whether or not the branch return cache (or the branch predictor in the SMT case) is valid;

5) BRCW – this field indicates whether or not the latest branch caused a write into BRC or the branch predictor. In the SMT case, every branch effects this bit;

6) BRCH – this field indicates if the latest branch has a BRC or a branch predictor hit;

7) MP – this field indicates whether or not the latest conditional branch verified was mispredicted;

8) BPMode – the most interesting field that controls branch prediction algorithms: 00 – 2-bit counter, in the SMT case – Global Indexed dynamic prediction, scheme; 01 all conditional branches are predicted not taken, in the SMT case – Global Mixed, scheme; 10 – all conditional branches are predicted taken, in the SMT case – Local, scheme; 11 – forward conditional branches are predicted not taken and backward conditional branches are predicted taken, in the SMT case – McFarling’s, schema;

9) BPState – this field contains the new 2-bit state for a conditional branch after it is verified. It is also used to hold the 2-bit state to read/write when a branch prediction table read/write operation ix executed;

10) BPIdx – this field contains the index to the Branch Prediction Table (BPT) for BPT read/write//initialization operation or the branch predictor index in the case when the SMT mode is enabled. The upper six bits of the BPIdx field contain the line address for BPT line initialization operations; the lower three bits of BPIdx are ignored;

11) BPOp – 00 read, 01 write, 10 initialize to “strongly not taken” mode, 11 initialize to “strongly taken” mode. The Performance Counter register now is more useful, provided that the CPU uses the performance counters for monitoring (for more details see the “Annotations vs Performance Counters” section).
Chapter 4. Developing SMT Topsy

- *Section 4.1* discusses slow interrupts.
- *Section 4.2* describes the process of loading executable files into Topsy.
- *Section 4.3* describes the basic Topsy architecture.
- *Sections 4.4-4.7* discuss SMT Topsy modules and structures.
- *Sections 4.8,4.9* illustrate changes required to turn Topsy to SMT Topsy.
- *Section 4.10* describes exception handling in Topsy and SMT Topsy.
- *Section 4.11* explains SMT command shell

### 4.1. Slow Interrupts

The main characteristic of the slow interrupt/the bottom half is that it is the part of an interrupt handler that is not required to be executed right away. In a case where the sequence of interrupts is executed, it may happen that the slow interrupt will not be executed at all. Any interrupt handler is divided into a top half and a bottom half. The top half of the interrupt is the part that is always executed immediately after an interrupt occurs. At that time the bottom half is deferred. Some interrupts may not have bottom halves at all. This type of functionality is usually achieved by making bottom and top halves separate functions and treating them differently. Usually the top part of the interrupt decides whether the bottom half will need to run. Consequently, the top halves include the set of operations that cannot be deferred. The main reason for implementing this additional functionality is to minimize overall interrupt latency. In addition, the idea behind the slow interrupts is to make them interruptible. Therefore, while a fast interrupt is processed, any incoming interrupts, either slow or fast, must wait. To handle these other interrupts as soon as possible, the kernel defers as much processing as it can to the bottom halves. There is another reason for implementing additional functionality. The complete discussion of this reason is beyond the scope of this work, and here the associated process briefly is mentioned briefly.

At the lowest level, the interrupt controller chip disables the specific IRQ that is being serviced. This allows the kernel to execute the top half. Since any peripheral in the system is
expected to work in the mode that is close to real time, the execution should be made as fast as possible. There is still a reason for separating top and bottom halves because some parts of a handler (its bottom half) do not have to be executed on every single interrupt instance. This separation implies that the top half (or any other piece of the kernel code) marks a bit in the bottom half and, by this, indicating that the bottom half must be performed. Once the bottom half is marked it stays so until the kernel executes it. Interrupt actions structure is a typical data structure for the slow interrupts processing. [11]. The structure defines an action the kernel should take upon receiving an IRQ. There are three types of interrupt actions structure. The first one consists of the following:

1) handler – usually implemented as a pointer to a function that takes some action in response to the interrupt;
2) flags – this part is not applicable to current discussion; for more details see [11];
3) mask – this field is architecture dependent and is used by devices to extract additional information while processing the interrupt;
4) name – a name associated with the device generating the interrupt; in a case when more than one device shares the IRQ it helps to distinguish the interrupts;
5) device id – a unique id for the device type – usually assigned by the manufacturer and recorded here;
6) next – pointer to the next member in the linked list of the IRQ actions structures;
7) typename – a human-readable name of the controller;
8) startup – matches the events from the given IRQ with the associated controller;
9) shutdown – disassociate the event from the given IRQ from the controller;
10) handle – handles a single interrupt on the IRQ supplied to the function.

Another interrupt actions structure that also may be included in the previous one, describes the interrupt and usually contains the following members:

1) status – an integer, which bits are used to mask the status flags of the given interrupt.
2) handler – a pointer to a special structure that will be discussed below;
3) action – pointer to IRQ actions structure; usually contains just one member, except for the case when the IRQ is shared;
4) depth – IRQ reference counter, in terms of Windows; represents the number of current users of this IRQ description structure.
This member is used to ensure that the IRQ is not disabled while servicing an event that is in progress. Applicable to this structure are the status flags that are usually called “IRQ line status” and contain the following set: active – prohibits another interrupt to enter the handler; interrupt critical section, disabled – the same as active, but only applies to the disabled interrupts; pending – determines if the interrupt is pending and, if so, the kernel invokes the handler after enabling the associated interrupt; reply – the interrupt has been replied, but not yet acknowledged; autodetect – the IRQ is autodetectable. The listed above members completely describe the status of the IRQ, therefore there is no need to keep other members in the linked list of the IRQs. The only noticeable exception to this situation is when an IRQ is shared by several devices.

The final structure type deals specifically with bottom halves and contains the following members:

1) mask count – an array that tracks nested pairs of enable/disable requests for each bottom half. These requests are the direct result of the call to either enable_bottom_half or disable_bottom_half functions. These functions are analogous to the AddRef and Release functions in Windows. They increment and decrement mask count, accordingly. When the counter reaches zero, all outstanding disables have been matched by an enable, so the bottom half is re-enabled;

2) mask and active – these two members control if the bottom half gets run. They have a single bit dedicated to bottom half, which is manipulated by a mark_bottom_half function. Regardless of whether or not it was previously marked, any bottom half can be turned off entirely by clearing the corresponding bit in the mask member (for more details see the discussion on the enable_bottom_half and disable_bottom_half functions);

3) bottom half base – an array of pointers to the bottom-half functions;

4) symbolic names enumerator that associates a symbolic name with each of the bottom halves the kernel uses.

For example, to mark the timer interrupt, the kernel makes the following call: mark_bottom_half (TIMER_BH). Now a typical set of functions that allow linking and unlinking actions and interrupts will be described. Init_irq – initializes IRQ handling. The function sets up the interrupt descriptor table and sets the cascade interrupt and the FPU
interrupt depending on the supported architecture. It also fills in the IRQ descriptor array. For MIPS architectures it means initialization of ISA-based IRQs. The function sets up the interrupt descriptor table.

In MIPS architectures every hardware or software interrupt that occurs on the MIPS-based system is assigned a number, which is used by the CPU as an index into the interrupt descriptor table. The corresponding entries in the table are starting stack addresses of kernel functions to jump to when the interrupt occurs. This function is only needed during the kernel initialization. In Topsy this function is implemented as part of do_topsy_bootstrapping function.

Setup_SMT MIPS_irq function fills out the IRQ actions structure with an appropriate action for the given IRQ. This function will be used in order to setup the timer interrupt. This function also sets a shared flag in all actions in the array (basically, the kernel makes sure that any action may be accessed by one or more users). After this the function verifies if the IRQ may be shared with all existing IRQ actions already on this IRQ. [11] provides one solution that is based on the following fact: after the first action in the linked list of actions, any other actions are only allowed to enter the list if they agree to share their IRQs with the first action. Therefore, if the first action shares its IRQs, then any other action in the list will also share. Otherwise, if the first action does not share its interrupts, the entire list contains only one entry.

Request_irq – this function creates an IRQ actions structure from the supplied values and adds it to the list of IRQ actions structures for the given IRQ and fills it out with all necessary information. Free_irq is the inverse function of Request_irq. This function iterates through the list of the IRQ description structures, takes the element, where device ID matches its input argument, detaches this element from the list and de-allocates its memory. If there are no more actions associated with this device, the device is shut down.

Probe_irq_on implements a significant part of the kernel’s IRQ autoprobing. The basic idea is to find all the devices that are turned off (do not have any associated actions), turn them back on, filter out the devices that have generated bogus interrupts and returns the magic number that is used by a Probe_irq_off function as an authorization key. The Probe_irq_off function checks if its internal magic number matches the one generated by the Probe_irq_on function. If the function was invoked by any other logic, the supplied input
argument most possibly will be invalid. If the magic numbers match, the function then loops over all IRQs, searching for any devices that responded to the caller's probe. The special flag in the structure indicates that there is a pending interrupt waiting to be served (it is not a bogus interrupt, since all of those were detected and successfully removed by the Probe_irq_on function). The count of IRQs is incremented and the number of the first found IRQ is saved. Whether or not a particular IRQ is successfully autoprobed, the autodetect flag is dropped, and the handler is shut down again. If more than one IRQ was successfully autoprobed, the function returns with a negative number. Otherwise, returns either 0, or the number of the first successfully autoprobed interrupt.

At this point, the actual interrupt handlers will be discussed. A typical interrupt handler is a very simple assembly function. Linux uses a BUILD_IRQ macro to identify the jump address from the interrupt symbolic name, pushes the return address onto stack and then jumps to the common interrupt address. The common interrupt is also a very simple assembly routine that was introduced to decrease amount of written code (and the number of bugs introduced by the code). This function does some system call-associated work and calls the do_IRQ function that calls code that is specific to the interrupt controller chip. The do_IRQ function also handles any active bottom halves. The bottom halves are serviced in the following three cases:

1) when deciding which process should get the CPU next;
2) when returning from a system call;
3) just before returning from any interrupt.

Only one bottom half is allowed to run at a time. The kernel ensures this property by acquiring a lock for every ready-to-run interrupt. If the lock has been successfully acquired, the bottom half runs. It is necessary to mention at this point that the same logic is applicable to servicing of all the CPU exceptions.

The last part of bottom half discussion introduces the time interrupt case, the case that has been implemented in this work. The timer interrupt is associated with IRQ 0. First, the handler for the time interrupt is registered as the timer bottom half, using init_bottom_half function. When IRQ 0 is triggered, timer_interrupt reads the EPS register and then calls do_timer_interrupt. The do_timer_interrupt function, in turn, calls do_timer function that does the majority of work. This function updates the global variable that keeps track of the
number of system clock ticks since boot (or, precisely, from the moment when the interrupt handler was installed, which does not exactly match the time of the system boot). The function then increments the number of "lost" ticks, i.e. the ticks that have not been handled by the bottom half. The top half has been run (see exceptions for more details on timer exception), and so its bottom half is marked to be run as soon as possible. In addition to tracking the total number of timer ticks that have occurred since the last time the timer bottom half was run, it is necessary to know how many of these ticks occurred while in system mode. The global variable is incremented for a particular process if the process is running in kernel mode. If any tasks are waiting in the timer queue, the timer queue bottom half is marked as ready to run (see a short discussion on wait queues below). The timer bottom half mainly updates statistics (for example, it can update the used CPU time for every running context in the SMT system). It also gets the number of timer ticks that occurred since the last time the bottom half ran, and resets this count. If this number is non-zero, the kernel finds out how many of those ticks occurred in the system mode, updates system clock and updates process times. The bottom half is a great feature of the modern kernel. The only problem with bottom halves is that their number is limited to 32 (mostly for speed-related issues). However, Linux offers a simple solution to this problem by setting up to 32 wait queues associated with interrupts.

4.2. OS: loading executable files

Though SimOS/Topsy uses different strategy of handling of executable formats, there is a need for a very short discussion on how the kernel handles executable file formats. As the kernel is installed, a special linked list is created. This linked list keeps the list of binary formats. The iterator is called on this list finds the requested handler by comparing the magic numbers saved in the list with the magic number saved in the beginning of the binary file. Then the function extracts the address of the handler for the file format, and tries to execute it, passing the file as the input argument. The SimOS interprets the whole kernel as a binary file; therefore there is no need for a special handler to be maintained by Topsy. However, the
SimOS handles the kernel binary (combined with all user processes code) in the way described above.

4.3. Topsy: current architecture

Topsy (Teachable Operating System) is a simple, but yet complete, thread-based framework for studying the operating system concepts, as well as researching and implementing new features of the modern operating system. The Topsy can also be used as a basic operating system in research and development of new hardware, including CPU models. The Topsy has a strict modular structure, depicted in Figure 6 [12]):

![Moduler structure of Topsy operating system.](image)

Courtesy of George Fankhauser, Christian Conrad, Eckart Zitzler, Bernhard Pattner, Computer Engineering and Networks Laboratory, ETH Zürich

The kernel contains three main modules reflecting the basic OS tasks: the memory manager, the thread manager and the I/O subsystem. All kernel modules are independently implemented as threads and therefore preemptible. The communication mechanism is the same as in user threads: by sending and receiving messages, interaction between the kernel components takes place. This provides quick response to interrupts and automatic
synchronization between modules. The following sections will describe the initial Topsy architecture, as well as the necessary changes that are needed to be implemented in order to make Topsy the SMT-compatible operating system.

4.4. Switching to SMT: Bootstrapping process

The bootstrapping process is probably the only part of the project that does not require extensive work. The bootstrapping process of SMT Topsy is depicted on Figure 7:

![Bootstrapping process of SMT Topsy operating system](image)

Figure 7. Bootstrapping process of SMT Topsy operating system.

Courtesy of George Fankhauser, Christian Conrad, Eckart Zitzler, Bernhard Pattner, Computer Engineering and Networks Laboratory, ETH Zürich

The general guidelines are provided by [12]. Also, some additional information has been added to the guidelines:
1) Compilation of the kernel source code and the user program. The compilation has to be done by MIPS cross-compiler. For additional information on how to compile and setup the MIPS cross-compiler, its binary utilities and run-time libraries see the installation instructions. As the result of compilation, both kernel and user files in so-called “image” format are obtained, where all addresses are relative, and, if no additional work is done, will be placed in randomly selected continuous memory chunks. However, this is not appropriate, since the SimOS can only work with the images, for which it has known code and data segments as well as pre-defined addresses and names. Moreover, all the segments must be aligned to the paragraph boundaries. This alignment is accomplished by applying a special loader script (link.scr) to every binary file generated by the cross-compiler. Upon applying the script, each binary file’s code, all initialized and non-initialized data and stack segments are aligned and marked with pre-defined starting addresses. For more information on how to write a script see the documentation on the gnu loader [13].

2) In order to be loaded on any hardware, the image file must be converted to the specific format. As indicated by [12], the experimental board uses Motorola S-record format (namely, S3-format). For more information on S3 format see the s-record documentation. This is important to understand, that the Topsy image file must not be converted to the S-format, since the SimOS simulator only accepts the .ecoff-formatted files and .srec files. However, there is one file that must be converted into .srec format in order to produce the correct Topsy image file. This is a light-weighted “SMT user-loading” thread (hereafter, SULT). For more information on the definition and use of the SULT see the “SMT startup and dynamic loading” section.

3) Kernel itself has to be informed of this layout. There are two ways to accomplish this: 1) to patch the information into the kernel data segment; 2) to add another “meta-segment” at the address that is known by kernel. This meta-segment contains the sizes of every segment of the kernel and user programs, as well as starting address of the user program that is appended to the end of the meta-segment. To produce this segment map, the [Topsy usr man] introduced a special BootLinker tool. The tool reads the information provided by the gnu-size utility applied to the kernel and the user program. An important point is to remember is that the tool is used after the gnu-ld loader. Unless otherwise specified, the loader produces its output files in System V format (for more information on
System V formats see gnu-ld documentation). Therefore, the gnu-size utility has to be called with an option to produce the correct results. In addition the start address of the user program is appended to the segment map. The simplest way to identify this address is to produce the S3-record from the user program and read its last record (the last record in any S3-formatted file is marked “S7” and contains the starting address of the code). The BootLinker tool simply takes the last record of the user.srec file and copies it to the segment map.

4.5. **SMT startup and dynamic loading**

The user program startup address (see above) is a small _start assembler routine in file start.S that is responsible of setting up the Topsy boot address and calling the main function of Topsy.

The code is very simple and, like an init process in Linux, is mainly responsible for setting up the processor mode, kernel and user memory spaces and thread manager. Naturally, the kernel expects to find the __main symbol at this address. The main question now becomes: what needs to be done in order to be able to load multiple user threads, which is a necessity for the SMT architecture. Keeping this in mind, a correct loading strategy for the SMT operating system must be chosen, define a mechanism that implements the chosen strategy and finally implement and test it on a real hardware or some CPU simulator.

In order to choose a correct loading structure, the following schemas should be considered:

1) the CPU user space is initially broken into the number of sections, each of which is assigned to a particular process (or thread in this case). The bootstrapping code loads all threads at the beginning of execution. The thread is unloaded only in the case when the CPU completes its execution. The memory management of a particular thread is done based on the size of the assigned memory segment;

2) the special SULT mechanism is applied to the initial user threads. For the definition of the SULT see the previous section. The only responsibility of the SULT is to create the initial user processes along with their stacks and make the first scheduling decision.
For more details on how it is done in the SMT Topsy see the “SMT Thread Module” sections. Also, a number of special techniques allow the Topsy to be in control of the memory management for any particular thread at any given time as well as ability to communicate with the CPU in order to unload any finished, unused or faulty thread. Another set of algorithms defined in Topsy, combined with the SMT-related changes to the CPU architecture, allows the instruction feed from multiple threads at any given CPU cycle. For more information on definitions and implementation details see the “SMT Memory Management Module” section of this work.

Clearly, the first approach defeats the whole purpose of having the Topsy running on top of the architecture, since this is the OS that should be responsible for thread scheduling and management. Moreover, this approach would lead to a significant memory fragmentation, since there always will be some unused pieces of memory for every loaded thread, whereas some other threads might be suffering from the lack of memory at the same time. Also, the proposed schema does not allow the use of SMT, since the CPU still must feed the instructions from a single continuous memory segment (a single thread) at a time. All existing simulators cannot run an operating system and use the loading strategy described above.

4.6. SMT Threads Module

As it was explained in “Bootstrapping Process” section of this work, the Topsy starts the Thread Manager (analogous to Linux init) process as a part of its initialization. The implementation details can be found in file TMInit.c, function tmInit. The init procedure that the function follows is explained below.

In general, the initialization details for the SMT Thread Manager are not different from those for R4000 Thread Manager. For more information on tmInit, please refer to the “Init Procedure” in [12].

The first two arguments are initialized in mmInit (see “SMT Memory Management Module” section) and define the stack for the tmInit. The last parameter is the user code startup address, as it was described in the previous section. The function sets thread
identifiers to -1 for the Memory Manager and -2 for the Thread Manager. Both stacks are initialized and the contents of both thread structures are updated, including the setting of the context (register values). Both threads are added to the thread id list and the thread id hash list. Specific exception handlers are installed in order to kill faulty threads (for more information on exception handling see the “Exception Handling” subsection in “SMT Topsy” section of this work). Both threads are set to READY. Since more than one thread is allowed to be in the running state, the scheduler continues to make scheduling decisions until the hardware runs out of process slots (for more details on how the scheduler does it see the “SMT Scheduler” section).

The system clock is configured in the right mode and the clock interrupt handler is installed. The CPU are now being fed with the instructions from two threads (out of 8 that are in context) at a time, based on 2,8 schema. The basic idea of the 2,8 schema: based on some algorithm the best thread is chosen from the context (the reader can get more information about the CPU algorithms from the “Fetching Unit” section). The CPU takes up to four instructions from this thread and turns to the next best thread in the context. The second best thread also provides the CPU with up to 4 instructions. If any of the two threads generate an exception, the CPU jumps to exception vector, switches to the kernel mode and returns control to the kernel thread (the kernel thread is never unloaded from the CPU). Both CPU and kernel thread handle an exception without switching the entire context (8 threads). This work has already emphasized the fact that the SMT operating system operates on the same set of principals that any single-running-thread operating system. However, many principals were reviewed, as well as most of implementation details, in order to comply with the new structure and specifics of the SMT Topsy.

This section begins with discussion of the process IPC inside of the SMT Topsy. All system calls are based on the exchange of messages between the kernel (the Thread and Memory Managers) threads and user threads. Many of them require a reply from the kernel. The following functions represents the send and receive system calls: tmMsgSend and tmMsgRecv (file Messages.h).

The first function accepts the thread id of the potential receiver and the pointer to the message structure (see TM-related data structures). The function is non-blocking and returns
either TM_MSGSENDOK or TM_MSGSENDFAILED. The thread identifier may also be set to ANY, allowing broadcasting the message among all threads.

The second function attempts to read a message from the process message queue. The function accepts the thread identifier, from which it expects to receive the message. Similar to the first function, the thread identifier may take the value of ANY, allowing the listening thread to accept messages from any other thread. Unlike the first function, the tmMsgRecv function is blocking. Usually it blocks for the number of microseconds specified by the Timeout parameter. If there is a valid message for the listening thread in its message queue, the thread id of the sender is returned by From parameter, and the msg parameter points to the message structure. The kind of expected message is specified in the MsgId parameter. If the MsgId parameter is set to ANYMSGTYPE, the next arriving message (no matter which type) is taken. The function returns either TM_MSGRECVOK upon success or TM_MSGRECVFAILED upon failure.

The following is the discussion on the main thread management functions. tmStart system call creates a new user. It returns either TM_STARTOK when the thread has been created successfully or TM_STARTFAILED in case of failure. The function accepts the pointer to a new user process that starts as a new thread. By the analogues to the _beginthreadex function, it is possible to pass a single parameter, which could also be a structure, to a new thread via the third argument of tmStart. The fourth parameter is a string that associates an internal logical name for the new thread. On return, the first parameter contains the new thread id. tmExit is invoked when a user thread completes its execution and frees resources.

Similar to Topsy, the call to tmExit is implicitly done when the return statement in the main function of the thread is executed. The function uses a Topsy mechanism that prevents stack underflows. The tmYield system call allows the thread to yield the CPU to another thread, when non-preemptive scheduler is implemented. If there are no threads waiting to start their execution, the thread continues its execution. This function may also be used in the preemptive case to force a scheduling decision. A user thread uses tmKill function in order to kill another user thread. The id of the “victim” is supplied in the first parameter. On the other hand, if a user thread tries to kill a kernel thread it kills itself. For kernel threads
there are no restrictions, they may kill any thread running on the system. In any case, the function uses the wport and rport instructions to complete its work.

The following group of system calls exchange information about threads or do some lookups. These functions are tmGetInfo, tmGetFirst, tmGetNext and tmGetThreadByName. The first function of the group returns is the information on the thread, which is specified by the first input parameter, or the information on itself, if the value SELF is supplied. The second and the third functions help a user to enumerate all running threads on the system. A return structure is filled with the thread id, parent id, status and name. Some shell commands may be implemented using these calls. tmGetThreadByName searches for a thread called name, provided that the logical name was passed as name argument when the thread was started. tmGetTime system call is used to get the time of day. It is returned as seconds and micro seconds since January 1st, 1998, 00:00, UTC. All presentation and time zone issues are handled by the user. The default setting of the time slice on the SMT Topsy is 10 ms. An important point here is that the user should not confuse this system call with the time interrupt, which is handled in a completely different way. The analogous function to the tmSetTime Topsy function is not implemented.

The following is a description of the IPC communication process. Any system call causes a message to be sent to the kernel. The exact structure of these messages is presented below. Each system call causes the partial context switch on an invoking thread by executing the wport and rport instructions (this is not applicable to the kernel threads). Each system call is recognized via an identifier in the message structure. The list of these identifiers for the thread manager component is supplied in the file Messages.h. A dedicated function in the kernel (MsgDispatcher in TMIPC.c) processes syscall exceptions and copies the arriving message to the message queue of the destination thread. There is a single IPC mechanism in the SMT Topsy. The messages can be sent either between user and kernel thread, between kernel threads, or between user threads. All threads have a single fixed size FIFO queue for storing incoming messages, one queue per thread. The size of this queue is controlled by a constant MAXNBMSGINQUEUE (file Configuration.h). An attempt to place a message into a full queue causes the send operation to fail with TM_MSGSENDFAILED. A generic message type (Message) is created with the sender id (set and controlled by the kernel for security reasons) and the type of the message, which is set by the sending thread. Another
member of the Message structure is the union of all possible messages that can be exchanged in the SMT Topsy. Two additional structures are defined. If the kernel threads want to communicate with each other, the specific message structure of type KernelMessage has to be set. If two user threads attempt to establish a communication channel, the structure of type UserMessage has to be filled out. The SMT Topsy did not adopt the Topsy way of inserting and extracting messages. The original policy is a mix of FIFO and priority queuing. It is shown in Figure 8:

![Figure 8. Kernel thread messages queue organization of Topsy operating system.](image)

Courtesy of George Fankhauser, Christian Conrad, Eckart Zitzler, Bernhard Pattner, Computer Engineering and Networks Laboratory, ETH Zürich

The message queue is based on a fixed-size array of messages. The linking process of messages inside this array is controlled by two integers. Each of these integers refers to another array of integers. The value found in the second array is the index of the next available message. A value of -1 in the index array indicates that this is the last message of the list. Consequently, the reason for such implementation is to make sure that some messages are delivered to a thread before any others.

On the other hand, in the SMT Topsy there is a simple way to prioritize the delivery of messages. The kernel thread message queue structure of SMT Topsy is shown in Figure 9.
The SMT message queue is a linked list that can grow up to the value of MAXNBMSGINQUEUE. A thread has the possibility to request a particular message of type msgIdPending coming from thread threadIdPending. If such message already exists in the list, it is directly delivered to the thread. Otherwise, the receiving thread executes the context switch and the next time it is invoked, it will execute rport instruction. The source operand for this instruction is address of the next pending message of this type, or empty. In the last case, the function returns TM_MSGRECVFAILED. The message structure also contains the internal flag for the implementation of a finite state machine. A typical example of such a system call is the syscall. When this call is made, the calling thread usually waits for a reply carrying the return parameters.

With original implementation, it was not guaranteed that this reply was put into the first place of the queue, and it was possible that the calling thread received a message from another thread before the reply to the syscall arrived. To prevent infinite waiting, Topsy allows the expected messages to bypass the queue. The system call generates an exception that is caught by a corresponding kernel handler. Inside of the kernel message handler, the message is actually copied in the destination’s message queue via kSend function. The kSend function can only be invoked inside of the kernel. If the receiver’s message queue is full, or the destination thread does not exist, kSend fails. On success, kSend returns TM_OK. The function executes the wport instruction, if it needs to terminate a certain thread or for some other reason. A message may be received via the tmMsgRecv function. Similar to the sending of messages, a software exception is raised and the kernel checks whether a message is already available or not. If there is no message, the caller is put to sleep. On the other
hand, if there is a message, it is read with the kRecv. The kernel knows whether messages are pending by reporting the dedicated registers (mailboxes). The function returns both a sender id and a message. The internal logical structure of the Thread Manager includes interprocess communication task (IPC) and CPU scheduler. For more details on the SMT scheduler see “SMT Scheduler” section of this work.

At this point it is necessary to discuss the IPC in more details. When the SULT makes the tmStart function call, the tmStart system call message is sent to the message dispatcher (smtTMMain function, file TMMain.c). The SMT function supports the input arguments that are similar to the original Topsy implementation, as well as usual thread startup technique: the address of the function to be loaded, a parameter passed to the new thread, the information whether a user or kernel is to be created, the thread name, the id of the parent thread and a flag indicating whether a lightweight thread is to be created. In this case, if the value of the last parameter was specified as FALSE, the thread with minimal allocated resources (the size of stack is zero) is created. The Thread Manager allocates a new thread data structure via tmAlloc within the dynamic kernel memory. The vmAlloc call to the memory manager is performed to obtain an execution stack for the new thread. The initial size of the stack is a constant TM_DEFAULTTHREADSTACKSIZE. The stack is allocated in the kernel address space. A new thread id is generated and inserted in the thread id list and the thread id hash list. The thread id is reusable, but at any given moment every process is guaranteed to have its own unique id. The thread identifier of user threads is positive whereas it is negative for kernel threads (predefined values are TMTHREADID (-2) for thread manager, MMTTHREADID (-1) for memory manager, IOTTHREADID (-3) for the IO subsystem, and 2 for the SULT). The stack then is moved to the user address space if a user thread is to be created (this is always true in SMT case). If the move succeeded, the new thread data structure would be updated with the thread identifier, the PC (program counter), and the SP (stack pointer). The complete thread context is initialized (for more details see the next section). The top of the new stack is laid out in a predefined manner. This step is needed to be able to execute the tmExit function. In order to achieve this functionality, the upper stack words are initialized. The status of the new thread is set to READY.

The function call tmExit (or corresponding system call threadExit) performs the termination of a thread. This is done in the following manner: the thread identifier is
removed from the thread id list and the thread id hash table; the scheduler is informed about the completion of the thread; all former reserved memory resources are deallocated (stack, thread descriptor, and allocated memory); all threads waiting for a message of the exiting thread are notified. The general idea of automatic execution of the exitThread is as follows (it was taken from Topsy, but implemented in a different way): the ra register of the MIPS processor contains the return address. By setting this register to the beginning of the exit routine that is lying on the stack, this exit code is executed whenever a return (jr ra) in the program code (of the corresponding thread) occurs. Initialization of the ra register is done in tmStart. When a particular thread or the kernel executes a wport instruction, the PC counter of the exiting thread points to the start of the exit code on the stack. The first few instructions compute the start address of the exit message located above the exit code. This message is a tmExit message. The next two instructions are identical to the last ones in tmMsgSend. Therefore, all that needs to be done in order to properly exit the thread is to send a tmExit message to the Thread Manager. In order to kill a particular thread, the TM_KILL signal must be sent to the Thread Manager. When the thread manager recognizes this signal, the function tmKill is invoked. If this call is valid (in other words, if a user thread is not trying to kill the kernel thread), the function performs the same set of steps as tmExit. When a user thread wishes to yield the CPU, it invokes the function tmYield. If there is another user thread ready for execution, a scheduling decision is forced by a call to smtschedule. If no other user thread can be scheduled, the requesting thread can continue up to the next threadYield call or until it is preempted by another thread with higher priority (that is by kernel). The function is executed in exception context. It means that when the low-level message dispatcher handles a tmMsgSend system call exception it catches yield messages (TM_YIELD) and invokes directly threadYield. Messages with id TM_YIELD are not delivered to the Thread Manager message queue and therefore not dispatched in tmMain by the function tmMain. The reason for special handling of this message is the necessity of context switching after a scheduling. Since context switching can only be done in exception mode, threadYield also has to be executed in exception mode. Though the scheduling decision is made immediately in the exception mode, the thread does not necessary gives up the CPU, as it was described above.
In this paragraph more implementation details on only two functions will be provided: smtSchedulerRunning and smtschedule. The first function now returns an array of pointers to the Thread structures, along with the number of tasks that are recently in RUNNING state. In order to implement this function, the running member of the Thread structure from a single pointer has been changed to the Thread structure to the array of such pointers. The size of the running member of the Thread structure is defined by the global constant CONTEXT, which is currently set to 8.

The second function just invokes the schedule function the number of times. This number is identified either by the increasing numbers of running contexts, which is returned by smtSchedulerRunning or the saturation of the attempts counter that may reach the value SMT_RESCHEDULE_ATTEMPTS. This value is now set to 10, but this number is certainly a subject for a deep research.

### 4.7. Demand Paging in SMT Topsy

This section explains how to organize a real demand paging in SMT Topsy. This section is theoretical and is not a part of the development. The beginning of this section will provide short definitions for virtual memory, swapping and paging and will describe the functionality of the memory unit. Then demand paging related data structures will be considered. The end of this section will provide a description of the user and kernel spaces from the demand paging point of view. The next section represents real memory management implemented in this work. The term “virtual memory” has dual meaning. First, this term provides the uniform memory view to the process. Second, the term stands for the technique of providing processes with their layout in memory. In the second case, a process does not know about its loading address. For any process, its starting address is zero. A computer systems did not implement virtual memory and only could move and entire process to or from the disk. The technique is called swapping, since one process is swapped for another.

Different sources pointed out that swapping technique also has its limitations. First, it requires an entire process to fit into memory simultaneously, so swapping becomes useless in
the case when the user wanted to run a process that required more storage than there was RAM installed on the system. Second, swapping could be inefficient. When it is completed, it has to swap out the entire process simultaneously, even if this involves swapping out an entire process when only a single byte is needed. Similarly, it has to swap in the entire process in order to execute only a small portion of the swapped-in application code. The invention of paging system solved all the above problems. Paging divides the system’s memory into smaller pieces – pages – that can be moved to and from disk independently.

Paging system has more bookkeeping overhead than swapping, because the number of pages can exceed the number of processes, but a lot of flexibility is gained in exchange. A paging system is faster, since there is no need to move the whole process to or from the disk. Pages usually have a fixed size on a particular platform. On MIPS CPUs, a variable-size paging system is available. Any computer system has the following address spaces: physical address space, linear address space and logical address space (also known as virtual address space). Physical addresses are the real hardware addresses available on a system. Paging can move a process (entire or just pieces) into and out of different regions of physical memory.

Logical address space is the set of addresses any process oversees during its lifetime. All logical addresses start at zero and extend to 3GB in 32-bit mode. Even though every process has the same logical address space, the corresponding physical addresses are different for each process, so no process address spaces are ever interleaved. Linear addresses are produced as the result of addition of a particular address to the base loading address. The linear address is converted to the physical address, which is a real address in the system’s RAM. Translating logical addresses to physical addresses, and vice versa, is the partial responsibility of the kernel and the hardware memory management unit (MMU). The kernel instructs the MMU which logical pages map to which physical pages for each process, and the MMU carries out the actual translation when the process makes a memory request. When an address translation is not possible (the supplied logical address is invalid or the logical-to-physical translation does not exist), the MMU signals the kernel by generating a page fault exception, which is handled by an appropriate exception handler of the kernel. The MMU is also responsible for enforcing memory protection. On MIPS CPUs the process of resolving a linear address (or logical address) to a physical address is a three-level procedure. The linear address supplied by the process is broken into four pieces: a page
directory index, a middle directory, a page table index, and an offset. A page directory is usually an array of pointers to page tables, and a page table is an array of pointers to pages. Therefore, resolving an address involves finding a middle directory by its corresponding page directory. A middle directory entry points to a page table entry, which corresponds to a page, and the offset gives an address within the page. This scheme is especially useful in 64-bit architectures by eliminating each of the following: enormous page tables, enormous offsets, enormous page directories, and/or all three together. The memory layout of a typical process can usually be represented as a following structure: virtual memory area (VMA), virtual memory operations (VMO), and memory management (MM). The VMA structure represents a single continuous memory area used by a process. Two VMAs never overlap – for a given process, an address is covered by at most one VMA; an address the process has never referred to in any way will not be in any VMA. Two VMAs may be discontiguous, that is, the end of one does not have to be the beginning of the other. Two VMAs may have different protections. Two VMAs that have different protections are managed separately.

4.8. SMT Memory Management Module

The SMT memory management scheme follows the general Topsy outline. Above the hardware abstraction layer there are three main modules: the message dispatcher, the management of virtual memory regions and the management of kernel heap memory. The message dispatches are implemented as a simple endless while loop, and an associated thread runs forever on the CPU model. The message dispatcher waits for the next message for the memory manager, decodes it and invokes the corresponding functions in the virtual memory manager. The message can also be generated by either the thread manager itself (for example, to kill the thread or to yield the CPU to another one), or by the CPU via exceptions. Yet another type of message is the system call. The exported routine mmMain is the main routine of the memory manager is called from the thread manager to start the memory kernel subsystem. Before this is done, all internal data structures must be initialized by the function mmMain. The following discussion will provide more implementation details on this function.
The hardware abstraction layer contains two modules: and error handler and the memory mapping functions. The error handler is responsible for handling of exceptions like address error and bus error. In this case the thread that caused the exception is terminated without complete context switch. For more details on the CPU exceptions and low level hardware dependent exception handling see the "Exception Handling" subsection of this section. Below more implementation details on the hardware independent exceptions handling will be provided. The memory manager needs to communicate with other (user or kernel) threads. Therefore the system calls tmMsgSend and tmMsgRecv are used. For exception handling also two other functions are required: smtTMSExceptionHandler and kSend. The first routine installs an exception handler for special exceptions. The second routine is necessary to send a message to the thread manager while handling exceptions. Since, in the case of an unrecoverable error a thread has to be terminated, a kill signal is sent via kSend to the thread manager (tmKill must not be used, since it generates a system call itself). The thread manager exports all functions or messages. The functions exported by the module mmHeapMemory are: hmAlloc, hmFree, and hmInit. The first function allocates a piece of internal kernel heap memory. It accepts the desired size of memory in bytes and returns the pointer to the allocated memory. If any error occurs HM_FREEFAILED is returned, otherwise HM_FREEOK. The second routine reserves a kernel heap memory of the desired size (the size is internally rounded up to a multiple of the page size). The start address is returned in the variable specified by the first parameter. The function returns either HMALLOCFAILED upon failure or HMALLOCOK upon success. The third function is responsible for the correct initialization of the heap data structure. The structure itself is a simple list. Every list entry has a field AreaStatus that contains information about the memory block lying between this list entry and the next list entry. The block is either allocated (HM_ALLOCATED) or is unused (HM_FREED). Upon successful call to the hmInit, the list contains two entries. The first is stored at the beginning of the heap area and the last is put at the end of the heap. The memory block between these entries is the total heap space that may be allocated by hmAlloc. To allocate a memory block, the Memory Manager first searches all list entries until a block of the desired size (or bigger) has been found. If this block is greater than required, it is split up into two blocks (an allocated one of the specified size and a free one of the remaining size) by inserting a new list entry. To free
memory, the Memory Manager invokes the hmFree. At this point, an earlier reserved memory block is set to HM_FREED. To avoid heap memory fragmentation contiguous free blocks are automatically merged; this procedure is only performed when hmAlloc was not able to find an area not big enough. To protect the memory for the simultaneous access by two or more threads, the kernel implements the locks.

The SMT Topsy manages virtual memory regions in the same way as any operating system. There are two types of virtual regions: user space virtual regions and kernel space virtual regions. For each address space two lists are maintained according to allocated and free virtual memory regions. The lists are implemented by using the abstract data type List. Each list entry contains the base address of the first page of the region, the size of the region, a protection mode, and the thread identifier of the owner.

The function mmInit works in the following manner: 1) the exception handlers concerning memory management are installed to catch bus, address and TLB errors. This is performed by the function mmInstallErrorHandlers in file MMError.c; 2) the TLB entries are set correctly by invoking mmInitMemoryMapping (file MMDirectMapping.c) and the user program is transferred to user space. The problem here is that the loader places the kernel code and data as well as user code and data in the segment kseg0. The kernel code and data must remain there, while the user code and data have to be moved to the segment kuseg; 3) the kernel heap is built. First, a virtual memory region for the heap is installed by calling mmVmGetHeapAddress (file MMVirtualMemory.c) and afterwards hmInit (file MMHeapMemory.c) is invoked; 4) the module mmVirtualMemory is initialized. In the kernel address space there are initially seven virtual memory regions: the boot/exception stack, kernel code, kernel data, kernel heap, memory manager stack, thread manager stack and a free region covering the kernel space that is left. The user space contains at the beginning a region for user code, a region for user data and a free region representing the unused user space, which can be allocated by vmAlloc.
4.9. SMT Scheduler

The section discusses the organization and design of the basic Topsy scheduler and the SMT Topsy scheduler.

4.9.1. Basic Topsy Scheduler

This section provides the detailed information on basic Topsy scheduler (function schedule() in file TMScheduler.c of Threads module). Figure 10 provides detailed description of the Topsy scheduling process:

![Topsy scheduler diagram]

Figure 10. Basic Topsy scheduler

Topsy utilizes the Round-Robin mechanism of process scheduling. All processes are organized in a single task queue. Previous and next members of each process are set to its siblings or the same process in the case when the process must be immediately rescheduled to run again. The basic MXS processor allows only one thread to be in running state at a time. The rescheduling process begins by allowing the available bottom halves to run. Every process keeps track of its own number of ticks. By freeing up the processor slot, the next available (not-blocked) process with the same (or higher priority) is moved to the running state.
The “goodness” value of a process is calculated by tmMain function of the Thread Manager. The Thread Manager based its decision not only on its own statistics, but also on the information provided by the Memory Manager (mmMain function). For example, if the process is blocked because of the page fault, its “goodness” value is decremented by the Thread Manager, and, as the result, this process does not run during the next available time slice.

The function RestoreContext() loads the scheduled process into the simulator. Among other parameters, it loads the current PC value into PC register of the MXS CPU.

4.9.2. SMT Topsy Schedule

This section provides the detailed description on SMT Topsy scheduler (function smtschedule() in file TMScheduler.c of Threads module). Figure 11 shows the SMT Topsy scheduling process:

![SMT Topsy scheduler diagram]

Figure 11. SMT Topsy scheduler

The SMT scheduler function has a self-explanatory name smtschedule. It usually depends on a particular process to be either Round-Robin, FIFO or OTHER. The last one
allows the process to give up the CPU and yield it to another thread. All processes are organized in a single task queue. Previous and next members of each process are set to its siblings or the same process in the case when the process must be immediately rescheduled to run again.

The SMT MXS processor allows up to 8 threads to be in running state at the same time. The rescheduling process begins by allowing the available bottom halves to run. The Round-Robin scheduler moves “exhausted” RR processes (up to 6), the ones that have used up their timeslice. Every process keeps track of its own number of ticks. By freeing up the processor slots, other RR processes with the same (or higher priority) have their chance to run. At the same time the scheduler replenishes the exhausted process’s timeslice. By contrast, the FIFO scheduler does not reset any timeslices, so those processes with higher priority do not yield the CPU when they run out of time.

As specified in [11], the smtschedule function is often is used when one task has decided that another task must be moved in or out of its running state. For example, if the hardware condition that a process was waiting for has occurred – the process switches its state. If the process is already in running state, it stays untouched. If the process was interruptible (waiting for a signal) and a signal arrived for the process, it is returned to its running state. In all other cases the process is removed from the run queue. First, the counter of user processes is initialized to 6. Then the function initializes the pointer to the first task in the run queue and loops through all the tasks in the queue. The best “goodness” value is identified (the discussion on the “goodness” of the task is not a subject of this work), and an appropriate process is put in the running state. Then the user processes counter gets initialized to the number of remaining unutilized processor slots. This effectively indicates the number of threads that the kernel can supply to the processor at this particular time.

Every initialized thread invokes the process of initialization of the user threads counter. If the value of the counter is 0 upon the initialization of the next thread, the processor is now running on its working set. It is not recommended, though for a programmer to attempt to initialize more than one thread per call. The problem is that supplying more than one thread to the processor may cause thread rejects. The reasons for those rejects may be unidentifiable at the kernel level. Therefore, the rejected threads may enter an identified state and will have to be killed by the kernel. The function then
recalculates the "goodnesses" of all remaining processes in the queue. For more details on context switching see the discussion of the SMT MXS architecture and implementation details. An important note here is that the kernel implements the context switch by executing rport and wport instructions, as it is described in the "Instruction Set" section. In the implementation of the calculation of goodness a general Linux procedure has been used.

This procedure is based on the number of ticks used by a particular process and the mode, in which the process was executed. However, with the SMT logic, the value of goodness is affected not only by pure software considerations, but also by the hardware (fetching and issuing units). The following is a short description of the process.

Goodness returns the value in one of two classes: under 1,000 and over 1,000. Values of 1,000 and up are assigned only to "real-time" processes (non-swappable processes, like message dispatcher or memory manager), and values from 0 through 999 are assigned only to user processes. The idle thread has a negative goodness counter. The function returns 0 in a case the process has yielded the CPU (the Yield bit is cleared first, because any process can yield only once, and it has done this). If it's a real-time process, the function returns a value placing it in the higher class. Otherwise, the code assumes processing of a normal task. It initializes the task's goodness to its current value (for more details see the task structure). The logic here is that the process that has already had the CPU for some period of time is less likely to be given a chance to run again. If the goodness is 0, the process's counter is used up, so goodness does not add in any more weighting factors. The scheduler moves to other processes.

The tmInit function associates each "empty slot" of the RUNNING queue with a particular PC register of the SMT MXS simulator. The number of the associated PC is passed as a parameter into the RestoreContext() function of each scheduled process. Therefore, the simulator can fetch instructions from several processes simultaneously.
4.9.3. SMT Topsy Scheduler Interface

The internal behavior of the scheduler can be described by four interface functions, which are SchedulerSetReady, SchedulerSetBlocked, smtschedulerRunning and smtschedule. Each function describes a transition of the states of a particular thread. The first function (BLOCKED – READY transition) represents the moment when a thread is ready to be scheduled and is moved from the BLOCKED queue to the end of the READY queue (according to the priority level of the thread). The second function (BLOCKED) is invoked when a thread is put to sleep at the end of the scheduler BLOCKED queue (the thread is waiting for a message). The third function is different from the one implemented by original Topsy. It returns the array of pointers to the Thread structure of the current running threads. The input parameter is the size of the returned array. It is passed by reference. The last function picks the next ready thread with the highest priority. As it was described above, the “goodness” of the next process to be run is executed based not only on the number of ticks the process used, but also on the numbers returned by the hardware. There can be more than one task in RUNNING state at any given time. The actual context switch (saving and restoring of context) necessary after each smtschedule call is done via the HAL functions SaveContext and RestoreContext. Each thread has its own context, which is defined by the register values at every point in time. When a thread is set from state RUNNING to BLOCKED or READY, the thread context (register values) must be saved in order to guarantee correct execution for the next time. The RestoreContext function sets all registers to the values that have been stored before. The smtschedule call is always followed by an RestoreContext call as a new thread becomes runnable. Therefore, the RestoreContext function ends with a jump instruction to the PC of the new restored thread context. The smtschedule function can also run in time interrupt. Similar to the original Topsy design, the SMT Topsy has three priority levels, which are kernel, user, and idle. For each priority level, there are two thread lists. The first one contains all threads in READY status, and the second one contains all threads in BLOCKED status. The Scheduler_t structure contains an array of ThreadPtr – an array of running threads, unsigned long running_size variable that indicates the size of the array of current running threads, and the priority queues.
4.10. Exception Handling

All types of the exceptions that SMT Topsy supports are listed in ExceptionId, all interrupts are listed in InterruptId (we do not include the discussion on interrupts in this work, except for the time interrupt). As it was described in the “SMT CPU Model”, whenever a common exception occurs, the CPU jumps to a predefined address according to the type of exception. For example, if a Reset happens, it jumps to the address stored in 0xbfc00000. An UTLBMiss exception (a virtual address in user space cannot be translated into a physical address because there is no entry in the translation look-aside buffer) causes a branch to the address specified in address 0x80000000. UTLBMiss exceptions are caught by the smtUTLBMissHandler. All other exceptions are handled by smtGeneralExceptionHandler (file TMhalAsm.S) located at address 0x80000080. The steps performed by the smtGeneralExceptionHandler are:

1) Save registers to be modified by the exception handler,
2) Set stack pointer and frame pointer of the freed thread on the exception stack,
3) Save context of the freed thread,
4) Call the specific exception handler,
5) Restore context of the current thread if identified by the exception handler.

A specific exception handler can be installed by smtTMSetExceptionHandler in TMhal.c. Since hardware interrupts are a special kind of exception a global interrupt handler (smtTMExceptionHandler in TMhalAsm.S) is installed by smtTMSetExceptionHandler. Similar to common exception handling there is a jump table that contains start addresses of specific interrupt handlers. Specific interrupt smtInterruptHandler handlers are installed by the special smtTMSetInterruptHandler. There are also two functions that are responsible for the initialization process of SMT Topsy. The function smtInitBasicExceptions located in TMInit.c and called by the startup code writes the correct start addresses of smtGeneralExceptionHandler respectively UTLBMissHandler to the address 0x80000080 and 0x80000000. In addition, for each exception or interrupt, an appropriate exception or interrupt handler is installed. The second function smtTMInstallErrorHandlers (TMErrror.c) registers two exception handlers: for hardware interrupts (smtHWExceptionHandler) and for the Syscall exception caused by the syscall instruction (smtSyscallExceptionHandler in TMhalAsm.S). The syscall exception handler is needed to implement smtTMMmsgSend and
smtTMMsgRecv. Also the clock interrupt handler is set (smtTIMClockHandler in TMScheduler.c) that only invokes smtschedule. The programming interface to the hardware clock includes two enumerators and two functions. The first enumerator ClockId specifies the timer, to which the function is applied. There are two timers CLOCK0 and CLOCK1. The second enumerator ClockMode specifies the clock mode of an appropriate timer. The clock is configured by a call to setClockValue with the timer identifier, the required period in milliseconds, and finally the clock mode. It is necessary to reset the clock each time an interrupt occurs with a call to tmREsetClockInterrupt. Furthermore, four exception handlers are installed to catch errors and handle faulting threads.

### 4.11. SMT Topsy Shell

SMT Topsy also inherits the original Topsy command line shell, which enables the user to start threads, kill threads and get information about threads. The available commands are start, exit, ps and kill. All threads that are registered in userCommands in file shell.c can be started or stopped. The function where the thread execution should start is specified by the name of the thread assigned in userCommands. The first parameter of the new entry into the array specifies the name, the second corresponds to the program to be called and the third parameter defines the argument passed to the thread to be created. It is also possible to enter several parameters in the command line. Per default the shell parses the arguments and writes them to argArray, which is, defined as a fixed size array of char pointers. By specifying argArray as parameter in the userCommands array a thread is able to take over more than one argument. Putting the “&”-sign after the name of the program will tell the start process to start the process in the background. After the “&” sign the Shell does not wait for a program to end and can process further commands. If the function has any arguments & causes a setup thread to run in the Shell. This setup thread copies the thread’s arguments into its stack to memorize them before starting the background shell function. The exit command instructs the shell to kill a current shell thread. The ps command reports the current process status. No arguments are needed. The resulting screen contains three columns. The first column reports the thread’s identifier, the second the thread’s state
(ready, running, or blocking) and the third is its name. The kill command kills any thread specified by its identifier. The identifier is taken from the result of execution of the ps command. SMT Topsy also runs the crashme program that is a remake of the famous Unix test program. This program attempts to crash the operating system by sending random messages to random threads to check the stability of the system. In addition, it stresses the memory and thread manager by allocating memory and starting threads until it gets an error back or the system crashes.
Chapter 5. Simulation Results

- Section 5.1 demonstrates the results of the simulation of the SMT MXS fetching unit and compares its performance with performance of the basic MXS fetching unit.
- Section 5.2 demonstrates the results of the simulation of the SMT scheduler and compares them to with performance of the basic Topsy scheduler.

5.1. SMT fetching unit performance

This section begins with a short discussion on the solution to the SMT fetching problem described in the “SMT MXS Fetching Unit” section of this work. Figure 12 shows the SMT MXS fetching process.

Figure 12. SMT MXS fetching process
The issue here is that each of the PC counter registers is loaded dynamically by SMT Topsy (a call to the RestoreContext function). However, since the ms_execute function is not functional, none of the PC counter registers can be dynamically re-initialized. To produce the data, the following solution was found in the course of this research: the use of the basic Topsy kernel code as the basic instructions base and its PC counter as the base PC counter register. When there is a need to switch the task, the base kernel PC counter is reloaded into the next available PC counter register, and previous PC counter register is set to 0 (SaveContext function simulation), as it is shown in Figure 13 below:

The maximum number of the PC counter registers is simulated by a random number generator.

Figure 14 illustrates the basic MXS Fetching Unit work cycle. The maximum number of instruction that can be fetched at a time equals to 4. Figure shows the SMT MXS
fetching process where at most 2 active threads are allowed at a time. It is important to notice that the maximum number of instructions (fetch width) in the SMT MXS Fetching Unit equals to 8. Figures 15, 16, 17 demonstrate the SMT MXS fetching process with the maximum number of active contexts equal to 2, 4 and 8, respectively. In figures, C1 represents the actual data series. However, the real comparison between the performance of the none-SMT and the SMT fetching units can only be made, assuming the same fetch width of the units (4 instructions per cycle). Figures 18, 19 below demonstrate the comparison and show approximately 60% decrease in the horizontal waste, resulting from the use of the SMT MXS Fetching unit, assuming ideal instruction flow of execution and the maximum number of active contexts equals to 4. In the real life, however, such gain in performance can never be reached and may even become 0% on different workloads and simulated configurations.

Figure 14. Basic MXS Fetching Unit performance
SMT MXS Fetching Unit (fetch width 8 instructions per cycle, 2 active contexts)

Figure 15. SMT MXS Fetching Unit performance (2 contexts)

SMT MXS Fetching Unit (fetch width 8 instructions per cycle, 4 active contexts)

Figure 16. SMT MXS Fetching Unit performance (4 contexts)
Figure 17. *SMT MXS Fetching Unit performance (8 contexts)*

Figure 18. *Comparing none-SMT and SMT MXS Fetching Units*

Where C1 series represent none-SMT MXS Fetching Unit performance, C2 series represent SMT MXS Fetching Unit performance.
Comparing MXS and SMT MXS Fetching Units (fetch width 4 instructions per cycle)

Figure 19. Comparing none-SMT and SMT Fetching Units

5.2. SMT Topsy scheduler performance

Figure 20 shows the results of the scheduling process, when none_SMT Topsy is run on the Intel architecture. Each process, including Memory and Thread Managers, can be either scheduled or not scheduled during a particular time slice (each process can be blocked on page faults). This logic should result in presence of empty slots on the graph. However, the basic Topsy architecture implements the IdleThread logic. The IdleThread cannot be blocked, and it is scheduled to run if there is no other READY process available on the system. Therefore, as it is shown in the Figure, each RUNNING tree turnaround (three time slices) is fully occupied.
Figure 20. Basic Topsy scheduling process

Figure 21 shows the results of scheduling process of the SMT Topsy running on the Intel architecture, with 6 uniform tasks loaded into the simulator. At any time, the maximum number of available "slots" in RUNNING queue equals to 8. In other words, if at least 8 processes are available in the system and no process are blocked on memory exceptions, the number of scheduled tasks has to equal to 8 during any time slice of execution (see Figure 21). Since there are two kernel level processes in the system (Memory and Thread Managers) that have scheduling priority, the maximum number of user threads that can be scheduled for execution equals to 6. However, as Figure 21 shows, during the most time slices, there is at least one user process that cannot be scheduled during the time slice. For example, during time slice 4 in the Figure, only 6 user tasks, Memory Manager and Thread Manager were scheduled for execution.
Figure 21. SMT Topsy scheduling process, uniform tasks

Figure 21 shows the results of scheduling process of the SMT Topsy running on the Intel architecture, with 6 none-uniform tasks loaded into the simulator. At any time, the maximum number of available "slots" in RUNNING queue equals to 8. In other words, if at least 8 processes are available in the system and no process are blocked on memory exceptions, the number of scheduled tasks has to equal to 8 during any time slice of execution (see Figure 22). Since there are two kernel level processes in the system (Memory and Thread Managers) that have scheduling priority, the maximum number of user threads that can be scheduled for execution equals to 6. During the most time slices, there is at least one user process that cannot be scheduled during the time slice. For example, during time slice 1 in the Figure 22, only 2 user tasks, Memory Manager and Thread Manager were scheduled for execution.
Figure 22. SMT Topsy scheduling process, uniform tasks

Figure 22 shows the results of comparison of the SMT Topsy scheduling processes running on the Intel architecture, 6 uniform tasks vs. 6 none-uniform tasks. Beginning at Sample 1 (time slice 504), only few none-uniform tasks remain loaded into the scheduler. Beginning at Sample 30 (time slice 534), when the last none-uniform task finishes, only the ThreadManager or the MemoryManager are scheduled by the simulator.
Figure 23. Comparing SMT Topsy scheduling processes, uniform and none-uniform tasks
Chapter 6. SMT MXS SimOS/Topsy Installation and Configuration

The SMT MXS integrated environment requires a few important components, which have to be installed prior to the installation of the environment: binary utilities and MIPS cross-compiler. The following sections describe the installation and configuration process of the SMT MXS integrated environment.

- Section 6.1 contains binary utilities installation instructions.
- Section 6.2 contains MIPS cross-compiler installation instructions.
- Section 6.3 contains SMT SimOS/Topsy integrated environment installation instructions.
- Section 6.4 contains important information on the configuration of the SMT SimOS/Topsy integrated environment.

6.1. Binary Utilities installation instructions

6.1.1. mount cdrom "xgcc distribution"
6.1.2. copy binutils_291.tgz to /usr/local
6.1.3. tar zxfv binutils_291.tgz
6.1.4. rm binutils_291.tgz
6.1.5. cd binutils-2.9.1
6.1.6. ./configure --prefix=/usr/local --target=mips-idt-ecoff
6.1.7 make
6.1.8 make install
6.2. **MIPS Cross-compiler installation instructions**

6.2.1. mount cdrom "xgcc distribution"
6.2.2. copy gcc_281.tgz to /usr/local
6.2.3. tar zxvf gcc_281.tgz
6.2.4. rm gcc_281.tgz
6.2.5. cd gcc-2.8.1
6.2.6. in file bd-emit.c: line 982 comment out
6.2.7. /configure --prefix=/usr/local --target=mips-idt-ecoff --with-gnu-as --with-gnu-ld
6.2.8. Copy /usr/local/include into /usr/local/mips-idt-ecoff/include
6.2.9. If needed, copy /usr/local/lib into /usr/local/mips-idt-ecoff/lib
6.2.10. If needed, copy /usr/local/src into /usr/local/mips-idt-ecoff/src (this is needed if the installation does not see the linux kernel).
6.2.11. make
6.2.12. make install

6.3. **SMT SimOS/Topsy integrated environment installation instructions**

6.3.1. Unpack the SimOS Runtime files. The default directory should be /usr/local/SimOS, if you don't want to change any relative paths.

   tar zxvf SimOS-runtime.tar.gz

6.3.2. Since SimOS is configured using tcl/tk scripts, you also need to install appropriate versions of tcl. The SimOS installation instructions stated that either tcl7.6 or tcl8.0 would work. However, this work recommends installing tcl7.6, since the tcl8.0 is not a backward compatible and is not recognized by some SimOS configurations.

   cd /usr/local/SimOS/tcl
   ln -s init.tcl7.6 init.tcl

6.3.3. Unpack Topsy-R4000.tar.gz (tar xvfz Topsy-R4000.tar.gz)
6.3.4. Do not change anything in init.simos. In /usr/local/Topsy/Makeconf change the first line from:

GNUMIPS=/usr/local/mips-idt-ecoff/bin/mips-idt-ecoff to:

GNUMIPS=/usr/local/bin/mips-idt-ecoff. The reason being: the last line of the gcc crosscompiler's Makefile moves the executable file into /usr/local/bin.

6.3.5. export TCL_LIBRARY=/usr/local/SimOS/tcl and put it into your .profile.

6.3.6. At this point, the installation cannot find a tcl archive file tcl8.0.a. In file /usr/local/SimOS/src/cpus/simos/Makefile.ALL line 95:

LIBTCL must be explicitly set to /usr/local/SimOS/build-files/lib/x86-shared/libtcl8.0.a

6.3.7. Now everything is ready to compile SimOS. Do not forget that the actual compilation must occur in /usr/local/SimOS/src (not in the SimOS root directory). To do this, the following make command line has to be entered:

make CPU=X86 SIMOS_DIR='cd ..'

6.3.8. Change to Topsy directory

6.3.9. Compile Topsy without any special flags (make)

6.3.10. Copy simos.exe from /usr/local/SimOS/src/cpus/simos/SIMOS_X86/ to

/usr/local/simos

6.3.11. You always run simos.exe from Topsy directory. Therefore,

 cd /usr/local/Topsy and run simos.exe by entering the following:

./simos. If you don't want to go inside of Topsy every time, just add the following line to your .profile:

export PATH=$PATH:/usr/local/Topsy. Do not forget to delete simos.exe from

/usr/local/bin after each compilation, or you can delete the "mv" line from the Makeconf file in SimOS installation.
6.4. SMT MXS SimOS/Topsy Configuration

The last section of this work explains how to configure the SMT MXS simulator. It is important to understand than none of the parameters listed below can be renamed in any future development. The reason for it is that, since the simulator is an integral part of the global SimOS framework, the listed parameters are shared by all CPU models and cache models. Therefore, if any of the parameters changes its name, all the CPU and cache models will have to be changed accordingly. The hardware parameters for SMT MXS can be set using a file called sim_ms_param.h. The following tables show all the parameters that are included in the file.

Table 7. Number and types of registers in SMT MXS simulator

<table>
<thead>
<tr>
<th>Name</th>
<th>Current Value in SMT MXS</th>
<th>Value suitable to simulate MIPS R10000</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_IREG</td>
<td>34</td>
<td>34</td>
<td>Number of integer registers</td>
</tr>
<tr>
<td>N_FREG</td>
<td>32</td>
<td>32</td>
<td>Number of floating point registers (in single precision)</td>
</tr>
<tr>
<td>FPREG</td>
<td>N_IREG</td>
<td>N_IREG</td>
<td>Offset to FP registers</td>
</tr>
<tr>
<td>HILOREG</td>
<td>(N_IREG+N_FREG)</td>
<td>(N_IREG+N_FREG)</td>
<td>Offset to HI and LO registers</td>
</tr>
<tr>
<td>FPCTL</td>
<td>(FPCTL+2)</td>
<td>(HILOREG+2)</td>
<td>Offset to FP control registers</td>
</tr>
<tr>
<td>TOT_REG</td>
<td>(FPCTL+2)</td>
<td>(FPCTL+2)</td>
<td>Total number of 32-bit registers</td>
</tr>
<tr>
<td>SP</td>
<td>29</td>
<td>0-34*</td>
<td>Index of stack pointer in reg file</td>
</tr>
<tr>
<td>LP</td>
<td>31</td>
<td>0-34*</td>
<td>Index of link pointer in reg file</td>
</tr>
<tr>
<td>CNDREG</td>
<td>32</td>
<td>0-34*</td>
<td>FP condition register</td>
</tr>
</tbody>
</table>

*SP, LP, CNDREG - can be set to any number between 0 and 34; however, they must be different from one another.
Table 8. Parameters related to register renaming and load/store operations in SMT MXS simulator

<table>
<thead>
<tr>
<th>Name</th>
<th>Current Value in SMT MXS</th>
<th>Value suitable to simulate MIPS R10000</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL_INST</td>
<td>352</td>
<td>352*</td>
<td>Reorder buffer size</td>
</tr>
<tr>
<td>IWIN_SIZE</td>
<td>96</td>
<td>96**</td>
<td>Instruction window size</td>
</tr>
<tr>
<td>LOAD_BYPASS</td>
<td>1</td>
<td>1</td>
<td>Bypass values from stores to loads</td>
</tr>
<tr>
<td>LDST_BUFFER_SIZE</td>
<td>32</td>
<td>16***</td>
<td>Load/store buffer size</td>
</tr>
<tr>
<td>LDST_BUFFER_MASK</td>
<td>0X1F</td>
<td>0X0F***</td>
<td>Mask for circular increment</td>
</tr>
<tr>
<td>PRESIZE</td>
<td>1</td>
<td>1</td>
<td>Support precise exceptions</td>
</tr>
</tbody>
</table>

* The active list in SMT R10000 serves as ROB, and it has $8 \times 32 + 100 = 352$ entries.
** SMT R10000 has 3 instruction queues, and each one has 32 entries.
*** MST R10000 has 16-entry address stack for use by non-blocking loads and stores.

Table 9. Bandwidth parameters of SMT MXS simulator

<table>
<thead>
<tr>
<th>Name</th>
<th>Current Value in SMT MXS</th>
<th>Value suitable to simulate MIPS R10000</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FETCH_WIDTH</td>
<td>8</td>
<td>8</td>
<td>Max fetches per cycle</td>
</tr>
<tr>
<td>ISSUE_WIDTH</td>
<td>8</td>
<td>8</td>
<td>Max issues per cycle</td>
</tr>
<tr>
<td>CACHE_WIDTH</td>
<td>2</td>
<td>2</td>
<td>Max loads+stores per cycle</td>
</tr>
<tr>
<td>WBACK_WIDTH</td>
<td>8</td>
<td>8</td>
<td>Max register writes per cycle</td>
</tr>
<tr>
<td>GRAD_WIDTH</td>
<td>8</td>
<td>8</td>
<td>Max graduations per cycle</td>
</tr>
<tr>
<td>THREAD_WIDTH</td>
<td>16</td>
<td>8*</td>
<td>Max number of threads to follow - Supports up to 8 predicted branches and up to 8 active ones.</td>
</tr>
<tr>
<td>MAX_Act_THREADS</td>
<td>8</td>
<td>8*</td>
<td>The number of active threads during speculative execution.</td>
</tr>
</tbody>
</table>

*SMT R10000 must have 8 entries in Branch Return Stack and conditional branches can be executed speculatively, up to 8-deep.*
Table 10. Branch prediction parameters of SMT MXS simulator

<table>
<thead>
<tr>
<th>Name</th>
<th>Current Value in SMT MXS</th>
<th>Value suitable to simulate MIPS R10000</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP_TABLE_SIZE</td>
<td>1024</td>
<td>512</td>
<td>Branch History Table Size</td>
</tr>
<tr>
<td>BP_RETURN_STACK</td>
<td>32</td>
<td>8</td>
<td>Branch Return Stack Size</td>
</tr>
<tr>
<td>BRANCHLIKELY</td>
<td>1</td>
<td>1</td>
<td>Support branch likely instructions</td>
</tr>
</tbody>
</table>

Table 11. Latencies of the various operations of the functional units of SMT MXS simulator

<table>
<thead>
<tr>
<th>Name</th>
<th>Current Value in SMT MXS</th>
<th>Value suitable to simulate MIPS R10000</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC_LATENCY</td>
<td>2 or 1</td>
<td>2 and 3*</td>
<td>Latency of primary data cache access (2 when ONE_PHASE_LS is defined). If ONE_PHASE_LS is defined, loads and stores must wait if there is a preceding store, and store must wait if there are any preceding loads or stores.</td>
</tr>
<tr>
<td>BRANCH_SLOTS</td>
<td>1</td>
<td>1</td>
<td>Number of branch delay slots</td>
</tr>
<tr>
<td>BRANCH_LATENCY</td>
<td>1</td>
<td>1</td>
<td>Latency of branch instructions</td>
</tr>
<tr>
<td>MULT_LATENCY</td>
<td>1</td>
<td>1</td>
<td>Latency for taken branch prediction</td>
</tr>
<tr>
<td>DIV_LATENCY</td>
<td>1</td>
<td>1</td>
<td>Latency for fall-thru branch prediction</td>
</tr>
<tr>
<td>FPADD_LATENCY</td>
<td>5</td>
<td>5(Lo) - 6(Hi)</td>
<td>Latency of integer multiply</td>
</tr>
<tr>
<td>FPMULS_LATENCY</td>
<td>34</td>
<td>34(Lo) - 35(Hi)</td>
<td>Latency of integer divide</td>
</tr>
<tr>
<td>FPMULD_LATENCY</td>
<td>2</td>
<td>2</td>
<td>Latency of FP add and subtract</td>
</tr>
<tr>
<td>FPDIVS_LATENCY</td>
<td>3</td>
<td>2</td>
<td>Latency of single precision multiply</td>
</tr>
<tr>
<td>FPDIVD_LATENCY</td>
<td>3</td>
<td>2</td>
<td>Latency of double precision multiply</td>
</tr>
<tr>
<td>FPSQRTS_LATENCY</td>
<td>12</td>
<td>12</td>
<td>Latency of single precision divide</td>
</tr>
<tr>
<td>FPSQRTD_LATENCY</td>
<td>12</td>
<td>19</td>
<td>Latency of double precision divide</td>
</tr>
<tr>
<td>FPABS_LATENCY</td>
<td>35</td>
<td>18</td>
<td>Single precision square root</td>
</tr>
<tr>
<td>FPNEG_LATENCY</td>
<td>35</td>
<td>33</td>
<td>Double precision square root</td>
</tr>
<tr>
<td>FPCVT_LATENCY</td>
<td>1</td>
<td>2</td>
<td>Absolute value</td>
</tr>
<tr>
<td>FPCVTSW_LATENCY</td>
<td>1</td>
<td>2</td>
<td>Negation</td>
</tr>
<tr>
<td>FPCVTDW_LATENCY</td>
<td>2</td>
<td>4</td>
<td>Latency of floating point convert</td>
</tr>
<tr>
<td>FPCMP_LATENCY</td>
<td>2</td>
<td>2</td>
<td>Latency of convert to fixed point</td>
</tr>
</tbody>
</table>

* Integer load/store latency is 2. FP load/store latency is 3. SIM MXS has only one variable, which is PC_LATENCY, and it treats integer and FP load/store instructions with same latency.
Chapter 7. Conclusion

This work defined the set of requirements to the SMT-compatible CPU and an operating system. It allowed to create complete specifications for SMT CPU and SMT-compatible operating systems. Using the SimOS and Topsy as a basic development framework, the simple, easily configurable SMT MXS fetching unit has been developed, which allowed us to compare none-SMT and SMT MXS fetching process, none-SMT and SMT Topsy schedulers. The work proved that the significant decrease in horizontal waste (up to 60%) can be achieved, when the SMT fetching unit is utilized.

The following problems have been resolved during development:

- MXS CPU model is fixed, and it is functional.
- The MXS CPU model is integrated into the basic SimOS/Topsy environment.
- The basic SimOS/Topsy environment provides full support for instruction and data caches: the instruction and data caches can be loaded with the appropriate segments of Topsy kernel.
- Binary utilities and MIPS cross-compiler are installed and used in the simulation.

At the same time this work opened a lot of possibilities for future development. Also, though this work has solved a large number of problems associated with the development, there are still few issues that need to be addressed.

The issues that need to be addressed before any future development take place:

- The ms_execute function has to be fixed in order for the instructions to execute on the simulator. The problems include a few missing files that are responsible for setting up the floating point unit mode. Also, a few problems with the management of the instruction window of the simulated CPU must be resolved.
- The design of the wport and rport instructions has to be verified and, if necessary, completed by addition of the necessary assembly routines.
- The new operating code, which corresponds to the CDI instruction, must be added to the development. If the ExecuteCDI function requires any assembly support, this support will also have to be added to the function.
• The complete framework must be tested against real workloads, including the use of the crashme utility. The last utility is needed in order to check the robustness of the whole system.

Once the complete robust SMT system has been completed, it then can be used in any future development, including the development of multiprocessor SMT-based system. The idea here is that the simulator can by itself support up to 32 different CPU models, both heterogeneous and homogeneous. Each of the processors can be individually configured to support the second level cache, whose size can also be individually defined per processor.

On the other hand, the simulator allows for treatment of any individual processor as if it were a peripheral device in the system. These capabilities allow the study of new heterogeneous systems, where any device may have its own real-time operating system and handle its own set of tasks under the main system CPU supervision.
# List of references


3. S. Parekh *et al.*, “Thread-Sensitive Scheduling for SMT Processors”.


