Modeling and Evaluation of Multi-core Multithreaded Processor Architectures in SystemC

by

Nicholas Ma

A thesis submitted to the Department of Electrical and Computer Engineering in conformity with the requirements for the degree of Master of Science (Engineering)

Queen’s University
Kingston, Ontario, Canada
August 2007

Copyright © Nicholas Ma, 2007
Abstract

Processor design has evolved over the years to take advantage of new technology and innovative concepts in order to improve performance. Diminishing returns for improvements based on current techniques such as exploiting instruction-level parallelism have caused designers to shift their focus. Rather then focusing on single-threaded architectures, designers have increasingly sought to improve system performance and increase overall throughput by exploiting thread-level parallelism through multithreaded multi-core architectures.

Software modeling and simulation are common techniques used to aid hardware design. Through simulation, different architectures can be explored and verified before hardware is actually built. An appropriate choice for the level of abstraction can reduce the complexity and the time required to create and simulate software models.

The first contribution of this thesis is a transaction-level simulation model of a multithreaded multi-core processor. The transaction level is a high level of abstraction that hides computational details from the designer allowing key architectural elements to be quickly explored. The processor model that has been implemented for this thesis is flexible and can be used to explore various designs by simulating different processor and cache configurations. The processor model is written in SystemC, which is a standard design and verification language that is built on C++ and that can be used to model hardware systems.

The second contribution of this thesis is the development of an application model that seeks to characterize the behavior of instruction execution and data accesses in a program. An application’s instruction trace can be profiled to produce a model that can be used to generate a synthetic trace with similar characteristics. The synthetic trace can then be used
in place of large trace files to drive the SystemC-based processor model. The application model can also produce various workload scenarios for multiprocessor simulation.

From experimentation, various processor configurations and different workload scenarios were simulated to explore the potential benefits of a multi-core multithreaded processor architecture. Performance increased with diminishing returns with additional multi-core multithreading support. However, these improvement were limited by the utilization of the shared bus.
Acknowledgements

I am extremely grateful to the many people who have supported me during this research. First, I would like to thank to my supervisors, Dr. Subramania Sudharsanan and Dr. Naraig Manjikian. This work would not have been possible without their support and guidance.

Financial support provided by Sun Microsystems, Inc. and Queen’s University is also greatly appreciated.

I would also like to thank the High Performance Computing Virtual Laboratory (HPCVL) at Queen’s University. HPCVL provided the necessary computing resources required to conduct this research. Due to HPCVL’s support, more time was spent on research rather than waiting on simulations.

Finally, I would like to thank my friends and family who gave me their encouragement and support throughout this research.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td><strong>Chapter 1</strong> Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Research Goals</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Contributions</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Organization</td>
<td>6</td>
</tr>
<tr>
<td><strong>Chapter 2</strong> Background</td>
<td>7</td>
</tr>
<tr>
<td>2.1 General Processor Architecture</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1 Instruction Pipelining</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2 Cache</td>
<td>10</td>
</tr>
<tr>
<td>2.1.3 Instruction-Level Parallelism</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Multi-core Multithreaded Processor Architecture</td>
<td>13</td>
</tr>
</tbody>
</table>
### Chapter 4 Model of an Application

4.1 Application Trace ........................................... 58
4.2 Instruction Type Model ........................................ 59
   4.2.1 Trace Analysis For Instruction Type ......................... 60
4.3 Address Generation Model .................................... 60
   4.3.1 Correlation Between Groups of Sequential Instructions ... 62
   4.3.2 Correlation Between Sequential Addresses .................. 64
   4.3.3 Address Generation Overview ............................... 66
   4.3.4 Trace Analysis for Address Generation ..................... 67
4.4 Model Variation Over Time .................................... 69
4.5 Parameter Selection .......................................... 72
4.6 Model Validation ............................................. 72
4.7 Multithreaded Application Modeling ........................... 84
4.8 Summary ..................................................... 85

### Chapter 5 Simulation and Experimental Results

5.1 Simulation Setup ............................................. 86
5.2 Experiments ............................................... 87
5.3 Experimental Results and Analysis ........................... 91
   5.3.1 Workload Experiments ................................. 92
5.3.1.1 Independent Core & Shared Threads Workload . . . . . . . 92
5.3.1.2 Independent Core & Independent Threads Workload . . . 95
5.3.1.3 Shared Core & Shared Threads Workload . . . . . . . 97
5.3.1.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . 99
5.3.2 Memory Experiments . . . . . . . . . . . . . . . . . . . . 100
5.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103

Chapter 6 Conclusion

6.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . 107

Bibliography 109
List of Tables

4.1 Sample of Clusters From FuzzyK Application .......................... 74
4.2 Instructions-Processed Ratio for FuzzyK Application and Model .... 75
4.3 Instructions-Processed Ratio for djpeg Application and Model ........ 82
4.4 Address Sharing for Multithreaded Application Modeling ............ 85

5.1 Modeled Cache Configuration .............................................. 89
5.2 Address Sharing for Multithreaded Application Modeling ............ 91
5.3 Modeled Memory Latency with Overhead ............................... 91
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Various Multithreading Schemes</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>MOESI Cache Coherence Protocol</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Block Diagram of Processor Architecture</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Block Diagram of Processor Core</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Cache Model Implementation</td>
<td>36</td>
</tr>
<tr>
<td>3.4</td>
<td>Implementation for Basic Pipeline Stages</td>
<td>37</td>
</tr>
<tr>
<td>3.5</td>
<td>Cache Access for Pipeline Stages</td>
<td>38</td>
</tr>
<tr>
<td>3.6</td>
<td>Pipeline Interconnect Channel</td>
<td>40</td>
</tr>
<tr>
<td>3.7</td>
<td>Pipeline channel Interface</td>
<td>41</td>
</tr>
<tr>
<td>3.8</td>
<td>Memory Request Interface</td>
<td>43</td>
</tr>
<tr>
<td>3.9</td>
<td>Pipeline Stage Memory Access Notification Interface</td>
<td>44</td>
</tr>
<tr>
<td>3.10</td>
<td>Internal Memory Channel</td>
<td>45</td>
</tr>
<tr>
<td>3.11</td>
<td>Internal Memory Communication Channel</td>
<td>46</td>
</tr>
<tr>
<td>3.12</td>
<td>External Core Communication Interconnect</td>
<td>48</td>
</tr>
<tr>
<td>3.13</td>
<td>External Memory Channel</td>
<td>49</td>
</tr>
<tr>
<td>3.14</td>
<td>External Memory Communication Interconnect</td>
<td>49</td>
</tr>
</tbody>
</table>
3.15 Thread Swapping Check Flow Diagram .......................... 52

3.16 Pipeline Core Structure Model ................................. 53

3.17 Process Core Constructor For Two Threads - Instantiations .... 54

3.18 Process Core Constructor for Two Threads - Connections .... 55

3.19 Processor Top Level Model ........................................... 56

3.20 Processor System Implementation For Two Cores .............. 56

4.1 Sample Trace Program Output ...................................... 59

4.2 Model For Generating Instruction Type ............................ 60

4.3 Trace Analysis Pseudocode for Instruction Type Generation Model .... 61

4.4 Two-State Markov Model for Address Generation [Sud04] .......... 61

4.5 Model For Generating Addresses in Terms of Groups .......... 63

4.6 Model For Generating Addresses in Terms of Spread .......... 65

4.7 Trace Analysis Pseudocode for Address Generation Model .......... 68

4.8 High-Level Representation of Application Model .................. 70

4.9 Application Model Flow .............................................. 71

4.10 Trace and Model Address Spread Probabilities for Linear FuzzyK Subsection .... 76

4.11 Trace and Model Address Spread Probabilities for Non-linear FuzzyK Subsection .... 77

4.12 Trace and Model Addresses for Linear FuzzyK Subsection .......... 79

4.13 Trace and Model Addresses for Non-Linear FuzzyK Subsection .... 80

4.14 Trace and Model Addresses for Most Frequent djpeg Cluster .......... 82

4.15 Trace and Model Addresses for Second Most Frequent Cluster .......... 83
5.1 Experiment Examples ............................................ 89
5.2 Independent Core and Shared Internal Addresses - Processor Performance . 93
5.3 Independent Core and Independent Internal Addresses - Processor Performance ............................................ 96
5.4 Shared Core and Shared Internal Addresses - Processor Performance . . . 98
5.5 Processor Performance with Varying Main Memory Latency .............. 101
Chapter 1

Introduction

Advances in semiconductor technology have fueled the evolution of system-on-chip design. What was previously manufactured on an entire board can now be fabricated within a single chip. The integration of processors and peripherals into a single chip is becoming more prevalent in embedded systems, desktop computing, and large-scale computers in consumer, business, and research applications. The new technology has enabled advances in functionality and performance, but has also increased the complexity for design and verification. Simulation is an important technique for architectural evaluation and hardware verification. Traditional functional-accurate, cycle-accurate, and register-transfer-level (RTL) simulation models require significant effort to design, and possible inaccuracies compared to hardware may not justify the initial effort. In addition, the simulation of complex systems using RTL or cycle-accurate verification methods is typically slow, thus making it unsuitable for running complex programs.

The current trend in industry is to focus on design and verification efforts before the RTL stage. Transaction-level modeling (TLM) is a technique that models systems at a
higher level of abstraction. At this level of abstraction, computation and communication
details are hidden from the designer, enabling system designers to focus on key architec-
tural issues. System designers can quickly develop these models, allowing for early verifi-
cation and exploration in order to prevent costly redesign during later development stages
[Pas04, CG03]. The high level of abstraction allows for the exploration of alternative ar-
chitectures and key architectural decisions can be addressed.

Moore’s law states that the number of transistors that can be integrated on a chip dou-
bles every 18 months. This trend has enabled processors to integrate more transistors on a
chip, enabling architectural innovations that increase performance and improve efficiency.
Advances in technology have enabled current processors to employ clock frequencies well
above 1 GHz. In contrast, memory speed has not been able to improve at the same pace.
Memory speeds have doubled only every six years, while processor frequency has dou-
bled every two years [Sun03]. This growing gap between processor and memory speeds
has prevented overall system performance from increasing at the same rate as the processor
clock frequency. To combat the diminishing returns of seeking to optimize single-processor
system performance, processor vendors have been developing products that include multi-
ple processors in order to increase throughput. Sun Microsystems, for example, has been
developing a chip multithreading (CMT) architecture based on multiple multithreaded pro-
cessor cores within the same chip [Sun03]. Multi-core multithreading design is a promising
approach for the future of processor design.

Processor development involves a large amount of modeling and simulation in order to
explore and verify architectural configurations. Software models have become increasingly
complex due to the inclusion of additional processor cores and multithreading support. In-
accuracies in the processor model may lead to poor design decisions and reduced performance once systems are implemented in hardware [TY03]. Detailed software modeling tools at the cycle-accurate level may require a significant amount of time to change, and the level of detail may not be suitable for high-level architectural exploration. In order to reduce complexity and development time, faster high-level architectural simulation can be used. One approach is to use modeling at the message level, corresponding to the movement of a block of data [Sud04]. This approach is an ideal way to verify and explore key architectural elements at early stages in the development cycle, such as the interconnect mechanism, the memory interfaces, and the memory system. This method is also flexible and models can be extended subsequently to lower levels of detail for further verification.

There are a variety of methods for processor simulation. One such method is execution-driven simulation. Execution-driven simulation models all data and operations but requires instruction processing and I/O to be fully emulated in order to simulate an application's execution. This method is very detailed and may not be suitable for high-level architectural exploration. An alternative method of simulation is trace-driven simulation. In this approach, a less-detailed function emulation of instruction processing is performed only once in order to generate a trace file containing the sequence of instruction and data addresses from execution, and this trace file is then used repeatedly as the input to the more detailed simulation. This method reduces the complexity, but the creation of instruction traces for long-running applications requires large amounts of storage space and a considerable amount of time to generate and process [ENSB03]. The use of either synthetic traces or statistical simulation attempts to reduce the demand for computing resources and simulation time. The synthetic trace scheme requires the analysis of an actual instruction trace
in order to reproduce a synthetic trace with characteristics similar to an actual program. The synthetic trace is generated from an application model during simulation, replacing the large instruction trace file. This method leaves the processor model unchanged when using either the synthetic trace scheme or an actual instruction trace. Alternatively, statistical simulation requires the analysis of the instruction trace file and processor response in order to reproduce similar processor behavior [ENSB03]. In statistical simulation, the processor model remains the same, except the detailed functional cache model is replaced with a statistical cache model. The method replaces the instruction trace file with a stream of probabilistic instruction events with no actual address information. During simulation, the modelled cache responses are based on probability rather than the simulated contents of the cache and memory over simulated time.

1.1 Research Goals

The research presented in this thesis has two goals. The first goal is to use SystemC to explore design options for multi-core multithreaded processors. A wide variety of design languages and associated software tools can be used to explore and verify hardware architecture decisions. SystemC is a relatively new library extension of C++ that provides the support for a new standard design and verification language and can enable different exploration methods. Early exploration of key architectural design elements can speed up the development time and prevent costly redesign.

The second goal is to explore methods to simulate long-running applications for single-threaded and multithreaded processors. Application trace files are large and time-consuming
to generate, and they only represent the execution on a single-threaded processor. To simulate an application on a multithreaded processor, an application model can be created to generate a synthetic trace for multiple threads with similar characteristics.

1.2 Contributions

The contributions of the research described in this thesis include a flexible model of a chip multithreading processor and a statistical model to represent an application. These contributions are described below.

The first contribution of this thesis is a transaction-level model of a chip multithreading processor that is implemented in SystemC. The model is flexible and can be used to explore variations in architecture, cache configuration, and memory timing. The model uses a high level of abstraction, enabling designers to focus on key architectural issues. This level of abstraction reduces the complexity and time required to develop and simulate the model, enabling key decisions to be made at earlier stages of the development cycle. The input for the model can either be an actual application trace or a synthetic trace based on an application model.

The second contribution is a model that seeks to characterize the behavior of instruction execution and data accesses in a program. The model distinguishes between different types of instructions and correlates sequential addresses with statistical probabilities in order to recreate an instruction sequence with characteristics similar to an actual program. The model can be used to synthesize instruction traces of single-issue applications, and it can be extended to synthesize multiprocessing instruction streams. The model can re-
place an actual application trace file, thereby reducing the storage space and execution time requirements for trace generation.

The first and second contributions can be used in combination to explore the effect of different processor configurations and different multiprocessor application scenarios on performance. This thesis presents results from such an exploration and draws conclusions on the circumstances for which multithreading across multiple processors provides performance benefits.

1.3 Organization

The remainder of this thesis is organized as follows. Chapter 2 discusses background that is relevant to this thesis. Chapter 3 presents details on the multi-core multithreaded processor architecture that is considered in this thesis and the implementation of a model for this architecture in SystemC. Chapter 4 discusses the application model created to simulate long-running applications. Chapter 5 discusses the simulation setup, the experiments, and the results. Finally Chapter 6 concludes with a summary of the thesis and directions for future work.
Chapter 2

Background

This chapter discusses background information that is relevant to the contributions of this thesis. First, general processor design is discussed because it forms the foundation for future processor architectures. Multi-core multithreaded processor design is discussed next, because it is one of the areas of future processor design and it is the main theme of this thesis. Finally, in order to develop current and future technologies, software modeling is discussed as a method that can be used to verify and explore architectural concepts and designs.

2.1 General Processor Architecture

Processor design has evolved over the years to take advantage of new technology and innovative concepts. However, the basic functionality of the processor has remained the same: to fetch and execute instructions. The processor itself is generally made up of registers, functional units (such as an ALU, logic units, and multiplexors), and control signals. Reg-
isters are used to hold relevant data for the processor. These registers may be used as the source and destination for the operations performed by the processor. The functional units within the processor are used to perform the required operations, and the control signals are used to set the correct data path between the source, the functional unit, and the destination, as well as set to the correct operation [HP03].

Many important architectural improvements have been developed over the years to improve processor performance. Some of the key innovations include instruction pipelining, enhancing memory performances through the use of caches, exploiting instruction-level parallelism (ILP), and more recently, multi-core multithreaded processing [HP03].

2.1.1 Instruction Pipelining

One of the key innovations to improve processor performance is pipelining [HP03]. Pipelining is a technique in which the execution of multiple instructions is overlapped. In a pipelined processor implementation, the processor’s functionality is divided into shorter stages. For a typical processor, these basic steps are: fetch an instruction from memory, decode an instruction, execute the operation or calculate an address, access an operand in memory, and write the results into a register. The processor must perform potentially all of these steps to completely process an instruction depending on its type. The critical path for timing within the pipeline stages is shorter compared to an unpipelined processor, which allows higher clock frequencies to be reached. The pipelined processor can overlap the execution of successive instructions, allowing for increased throughput.

The introduction of pipelining into the processor adds new issues that must be addressed by designers. There are some situations where an instruction cannot be executed correctly
because of structural, data, and control hazards. Both the instruction-fetch and memory stages may require memory access simultaneously, which will result in a structural hazard. To resolve this issue, separate instruction and data memory units are used. A data hazard occurs when there are dependencies between overlapping instructions. The operands of an instruction earlier in the pipeline may be the result of a later instruction that has not been processed by the write-back stage. In such cases, the pipeline must forward the correct data to the required stage. Unfortunately, not all data hazards can be solved by forwarding. A scenario may arise where data is read into the processor during the memory stage but is required in the execution stage by the immediately following instruction. This data, however, will not be valid until the end of the clock cycle. A stall must be introduced into the pipeline to ensure that the operand is valid so that the correct data can be then forwarded to the following instruction. Finally, control hazards may occur when the pipeline deals with branches. The decision to branch may not be known until further down the pipeline, but incorrect instructions may have been already fetched by the pipeline. The pipeline must flush these incorrect instructions and continue execution from the correct instruction. The fetching of incorrect instructions is a waste of processor cycles and results in a penalty that reduces the execution efficiency of the processor. Various techniques, such as 1-bit and 2-bit branch prediction, have been developed to reduce the frequency of branch misprediction and minimize the effects of their corresponding pipeline disruptions.

Deeper pipelines and improved transistor technology have allowed clock frequencies to increase significantly. However, deeper pipelines are more affected by costly pipeline disruptions, such as branch misprediction, because more incorrect instructions proceed deeper into the pipeline, resulting in more wasted cycles when flushing is required [HP03].
2.1.2 Cache

One of the main limitations on overall system performance is memory latency. Communication between main memory and the processor is slow compared to the execution speed of the processor. One of the methods used to minimize the effect of main memory latency is the use of caches. Caches are smaller, faster storage units placed in between the processor and a large main memory that attempt to reduce the effective latency for repeatedly accessed data. When data is found in the cache, it results in a cache hit that avoids access to slower memory modules. When data is not found in a cache, it results in a cache miss that requires access to slower memory to bring the data into the cache. Throughout an application’s execution, different memory locations will be accessed for instruction and data requests. However, these memory accesses are not random and two principles can be used to describe the memory access: temporal locality and spatial locality \cite{HPO3}. Temporal locality is the principle that if a memory location is accessed, it will likely be accessed again in the near future; instructions are repeatedly executed in loops and the same data location is read or written to multiple times. Spatial locality is the principle that if a memory access is performed, then adjacent locations will likely be referenced soon. This is true as an application generally executes a sequence of adjacent instructions in memory, or accesses a group of adjacent data locations. A memory hierarchy consisting of one or more levels of cache storage is used to take advantage of these principles to improve memory performance.

A cache can be designed with many different configurations based on parameters such as set associativity, cache size, and block size. The total cache size is split into $n$ indexed
segments based on the set associativity of \( n \) and the block size. The set associativity determines the number of data blocks that are associated with each index. Each block has additional status information associated with it called status and tag bits. The status bits are used to determine whether the cache contains the requested data. The tag bits are used to determine whether the cached data matches the request from the processor. To take advantage of spatial locality, the block size may be larger than a single word, allowing multiple adjacent words to be brought into the cache for a single cache miss, hence improving cache performance. When a memory request occurs, the address for the request is split into index, tag, and offset bits. The index bits are used to determine which cache segments the data might be located in. The tag bits from the requested address are compared with the tag bits in the cache at the appropriate index to check for a hit. On a hit, the requested word is then selected from the block of data using the offset bits. If the cache access is a miss, then the memory request is passed down the memory hierarchy to be processed by the next memory unit.

When a cache hit occurs, the data can be provided more quickly than having to access main memory. Because programs exhibit temporal and spatial locality, the frequency of longer latency main memory accesses is reduced with the use of caches. Nonetheless, the increasing gap in speed between main memory and processor hardware causes diminishing returns for performance optimizations related to caches \cite{Sun03}.

### 2.1.3 Instruction-Level Parallelism

One of the techniques to improve performance is to take advantage of instruction-level parallelism. Ordinarily, programs are written sequentially and instructions are executed
one after another. However, some of these instructions may not have any dependencies on nearby instructions. These instructions can be overlapped or their sequence may even be changed in order to take advantage of their independence. This property is known as instruction-level parallelism (ILP). To exploit ILP, designers have attempted to increase the number of instructions that a processor executes simultaneously. Techniques such as superscalar execution, out-of-order execution, and aggressive branch prediction exploit ILP and can increase the number of instructions executed in each clock cycle.

Superscalar architectures require additional hardware within the processor to exploit ILP. During a single clock cycle, the processor fetches multiple instructions that will be executed simultaneously. However, if the instructions do not meet certain criteria, the processor will not execute the extra instructions and the additional hardware will not be utilized effectively.

Out-of-order execution is a technique that exploits ILP. A typical out-of-order processor is divided into three subsections: fetch and decode, execute, and commit stages. The instructions are fetched at the start in order, but simultaneously in groups, and their results are typically committed at the end in order. During the execution stage, however, instructions may be processed out of order. The execute stage is comprised of multiple functional units with instruction buffers called reservation stations. Instructions from the fetch and decode stage are placed into the buffers of functional units. Once an instruction’s operands are available, the instruction is processed regardless of the original order in which the instruction arrived with respect to other instructions.

Techniques that exploit ILP have led to processor performance improvements. However, these improvements require more complexity, more transistors, and more power.
thermore, the performance improvement does not necessarily improve by an amount that is proportional to the increase in transistor count and power [MBH+02].

### 2.2 Multi-core Multithreaded Processor Architecture

Performance improvement is one of the main goals for designers in the computer industry. Previously, designers have focused on improving single-thread performance by increasing clock frequency, exploiting instruction-level parallelism (ILP), and enhancing memory performance through the use of caches. However, these improvements have added complexity and are subject to diminishing returns. In addition, the growing disparity between processor frequency and main memory access times has resulted in designers considering alternative approaches to processor design. Instead of single-thread performance, the focus of designers is on total throughput and overall performance by handling more than one instruction stream at a time with multiple threads and multiple processors.

Commercial server applications have become increasingly important in industry. These server applications process a large number of independent client requests that exhibit a large amount of parallelism. The parallelism can be exploited by thread-level parallelism, leading to higher throughput. Not only have commercial server applications tried to take advantage of this type of parallelism, but desktop applications have also sought to exploit thread-level parallelism. Hence, the focus on multithreaded processing is justifiable [KAO05, MBH+02].

Initially, symmetric multiprocessing (SMP) systems were proposed and implemented by the computer industry. These systems allow multiple threads to be processed in parallel
by multiple different processors. Each processor supports only one thread at a time, but with additional SMP processors, more threads can run simultaneously at a system level. However, this approach involves considerable external communication between processors through the interconnection network and the main memory. The next logical step for multiprocessing was to internalize the communication.

The current trend in industry is an approach that focuses on multiple cores and multiple threads per core within a single chip. Chip multi-threaded (CMT) processors combine support for chip multiprocessors (CMPs) and multithreading capability in each processor [Sun03, SA05]. CMPs allow multiple cores to reside within the same chip, enabling multiple cores to share resources and improve utilization. Multithreading allows the individual processor cores within a chip to execute instructions from more than a single thread at the same time, allowing useful computation to continue on the same physical core while another thread is idle due to a cache miss or a long execution event.

Despite advances in technology, it is not uncommon for the execution units to be utilized only 25% of the time across a wide variety of scenarios [KST04]. Thread-level parallelism has been introduced to increase utilization. Multithreading allows a single processor to execute instructions from more than one instruction stream at the same time. To software, it appears as if there are multiple processors present, and the operating system can exploit this feature in its management of threads. In hardware, however, the threads usually share pipeline resources within the same physical processor.

In industry, there are a variety of hardware allocation schemes for multithreading: coarse-grain multithreading, fine-grained multithreading, and simultaneous multithreading (SMT). Coarse-grained and fine-grained multithreading are both types of time-sliced
Coarse-grained multithreading allows for the execution of only one thread at any instant in time. The executing thread has complete control of processor resources until a long-latency event occurs. Such an event causes the processor hardware to switch to another thread rather than waste processor time. Hence, course-grained multithreading has also been called switch-on-event multithreading. By performing the thread swap, the processor core can continue to perform useful work, thereby increasing overall throughput. A long-latency event is usually determined by a set threshold; when an event latency exceeds the threshold, the context switch is triggered. A typical long-latency event is a cache miss. Hardware resources, such as architectural registers, are shared among threads to limit increases in chip area to support multithreading. Therefore, a thread-switching overhead for saving and restoring register contents is incurred to swap between threads. Figure 2.1(a) is
an example of the execution behavior of this hardware allocation scheme. IBM has implemented a coarse-grained multithreading scheme for the processor that is used in the IBM eServer pSeries Model 680 [KST04].

Fine-grained multithreading is similar to coarse-grained multithreading, in that the processor can switch to another thread during a long-latency event. However, in the fine-grained multithreading scheme, the previous thread’s data is not completely swapped out of the processor. This allows the threads to be swapped quickly and more frequently, hence, fine-grained multithreading is also referred to as interleaved multithreading. Because threads aren’t completely swapped out, some level of hardware replication is required for this scheme’s implementation, e.g., the architectural registers. Figure 2.1(b) is an example of the execution behavior of this hardware allocation scheme. Sun Microsystems uses a fine-grained multithreading scheme for the UltraSPARC T1 processor architecture [KAO05].

Simultaneous multithreading (SMT) is a multithreading scheme that also allows a processor to execute instructions from more than one thread. The difference is that instructions from different threads can be dynamically scheduled for different functional units at the same time. This allows execution units to be assigned to execute instructions from a single thread, or instructions from different threads. SMT is typically applied on superscalar processors that inherently require additional hardware for multiple pipelines. Figure 2.1(c) and Figure 2.1(d) are examples of a superscalar processor pipeline without and with simultaneous multithreading respectively. IBM has implemented the SMT scheme for its Power5 processor core [KST04], and Intel uses a SMT scheme that is called Hyper-threading [MBH+02] for its multithreading processors.
The level of multithreading on a processor is an interesting design decision, and a trade-off exists between complexity and performance. IBM’s dual-core Power5 design uses two-way SMT in each core. It is claimed that although more multithreading is possible, simulations show that the added complexity is unjustified [KST04]. The performance may even decrease due to cache thrashing, as different threads displace each other’s required data in the cache. On the other hand, Sun Microsystems has developed its CMT processor called UltraSPARC T1 with four-way multithreading [KAO05]. The T1 is a new implementation of the Sparc V9 architecture and supports a total of 32 threads. The 32-thread support is obtained by having eight processor cores, each with the ability to execute four threads simultaneously. Each core has its own modest L1 cache, and all eight cores share a large L2 cache. Multi-core multithreaded processing has become an important aspect in processor design. The major processor vendors, such as Sun Microsystems, Intel, IBM, and AMD, have all invested time and resources in the development of multi-core multithreaded technology [SA05]. This move towards multi-core multithreaded processor design represents a shift in design philosophy on how to improve overall performance.

2.2.1 Multiprocessor Cache Issues

In a multiprocessor environment, cache coherence is an important issue because multiple copies of the same data can be located in different processor caches simultaneously. It is important that the coherence of cache contents be maintained to ensure that proper values are accessed by software. The most popular coherence technique is called bus snooping, which requires that a shared bus supporting a common memory be monitored, i.e., snooped, by all cache controllers. Every bus request is checked to determine if further action is re-
quired. This action includes a cache controller providing the data from its cache in response to a bus request, or updating its own cache with new status information. These actions are determined by the cache coherence protocol.

A representative cache coherence protocol is the MOESI protocol [CGS97]. MOESI is a write-invalidate protocol in which each entry in a cache may be in one of five states: modified, owned, exclusive, shared, and invalid. The protocol can be viewed from two sides: that of the cache that initiates a memory request that appears on the bus, and that of each snooping cache that reacts to a request that appears on the bus. Using the MOESI state information, the system can determine which cache has the most up-to-date data and how the cache will respond to processor cache misses and snoop requests from the bus. Figure 2.2(a) represents the MOESI cache coherence protocol with respect to the cache that initiates a memory request. Figure 2.2(b) represents the MOESI cache coherence protocol from the view of the snooping cache. The states represent the situation for a given entry in the cache before and after a request. Transitions between states are determined by the result of cache and snoop requests.

The invalid state represents a cache entry that is not valid. A transition to the invalid state occurs when a store is performed by another processor to a location that is presently found in the cache in question. The corresponding request that is placed on the shared bus causes the copies of the data to be invalidated by all other snooping caches. The shared state indicates that more than one copy of the data exists in the L1 caches of the system. The modified state indicates that the cache entry has been modified, and is different from the data at lower levels in the memory hierarchy. The modified state occurs on any store hit or miss. Transitions from the modified state due to load or store misses require the modified
Figure 2.2: MOESI Cache Coherence Protocol
data to be written back to a lower level in the cache.

The more interesting scenarios in the MOESI protocol relate to the exclusive and owned states. The exclusive state indicates that only one copy of data exists in any of the L1 caches. It allows the processor to immediately write to that address and change its state to modified, without sending an invalidation request to the other caches, thereby avoiding unnecessary bus requests. The owned state, on the other hand, only occurs when a cache’s modified entry is requested by another processor due to a load miss. The cache controller provides the data, and changes its state to owned. This indicates that more than one copy of the modified data is in the local caches and has not been written back to main memory. If yet another processor requests the same data, there will be no ambiguity as to which processor should respond to the request: according to the protocol, the owner of the modified data will respond to the snooped request. The benefit of the owned state is that it removes the necessity to writeback to main memory when a modified entry is read by another processor; the writeback occurs only upon replacement in the cache with the owned state for that data.

Cache coherence plays a role in the performance of a multiprocessor system. To maintain cache coherence, processors must monitor the shared bus and if necessary, stall their own activity temporarily in order to address the needs of another processor. The overhead that arises from enforcing cache coherence reduces processor performance, but cache coherence is required to ensure correct execution.
2.3 Processor Modeling and Simulation

System designers generally use software models as an aid in the development process. These models are used to validate the performance and correctness of proposed hardware designs through simulation before they are built. Software models can also be used to compare various designs and configurations. These software models are typically implemented in traditional programming languages, such as C++, or hardware description languages (HDLs), such as Verilog. The models can then be used with appropriate workloads to simulate physical hardware. Software models are not as fast as their physical counterparts, but designers can create and test these models much faster than it would take to build the actual hardware. Any errors in hardware would require correction in the next hardware build, which could be costly and time-consuming. Software verification of hardware designs leads to faster development and earlier detection of errors.

Software models for simulation of hardware systems are designed with three issues in mind: performance, flexibility, and level of abstraction. For the first issue, the performance of the model itself is restricted by available resources for simulation support. Processor speed and simulation workload affect the runtime of a simulation. Consideration of available computing resources must be made to achieve simulation results in a reasonable time. A more complex model can be created to take advantage of additional computing resources if available. The second issue of flexibility is an important one for designers. The ability to make simple modifications, to vary the design more significantly, or even to use entirely different designs is important while exploring different architectures or configurations. The third issue on the level of abstraction relates to the level of detail with which the physical
hardware is modeled in software. A highly detailed model simulates all aspect of the hardware operation, regardless of its importance to a designer’s needs. Designers can choose a level of abstraction that suits their purposes. Generally, flexibility comes at the expense of detail because more complexity reduces the flexibility of the model. Variations in these three issues lead to many different approaches for modeling hardware designs in software.

Researchers usually focus on a simpler model with good simulation performance and a modular design for flexibility. A simpler model allows researchers to focus on primary design aspects, leaving out details that might reduce performance and flexibility. In industry, however, a more detailed model is usually created at the expense of flexibility to verify complex designs and minimize commercial risk. Performance of these detailed models is not usually an issue because commercial organizations are willing to invest in the necessary computational infrastructure that is required.

A variety of processor modeling tools exist. One of the more popular tools in academic research is SimpleScalar, which provides the infrastructure for simulation and architectural modeling [ALE02]. SimpleScalar simulates a program’s instructions and supports several different instructions sets. One of the benefits of SimpleScalar is that it can model a variety of architectures ranging from single unpipelined processors, to detailed dynamically-scheduled pipelined-processors with a memory hierarchy. SimpleScalar uses execution-driven simulation that requires that the processing of instructions and I/O be emulated in sufficient detail for application software. This approach simulates all of the actual data processing during the execution of a program, which can be important for some types of analysis. On the other hand, this approach might be too detailed and unnecessary for other types of analysis, and other simulation models or levels of abstraction may be more
appropriate.

In addition to SimpleScalar, some other processor and system modeling tools include SimOS [RBDH97], Simics [MCE+02], and General Execution-driven Multiprocessor Simulator (GEMS) [MSB+05]. These tools provide a full system simulation environment for different processor architectures and can run unmodified operating systems and applications. SimOS and Simics support several different processor architectures, and GEMS, which is based on Simics, details multiprocessor systems.

### 2.3.1 Transaction-level Modeling

In order to reduce complexity and simulation time, the design abstraction of a simulation model can be raised. Transaction-level modeling uses a high level of abstraction compared to that of the more detailed register-transfer level (RTL), and it can also be used to validate systems through simulation. In transaction-level modeling, the communication between components is separated from the implementation of those components. This division allows different communication and component modules to be used in order to explore a variety of designs while reusing created modules as much as possible. The component models are linked together by channel models through interfaces, and communication is modeled by the simulator placing transaction requests on channels through their defined interfaces. The low-level details of computation and communication are also hidden to decrease simulation time and complexity. Transaction-level modeling is not concerned with specific computational details or detailed communication between two components, but rather the exchange of data, or an event representing the interaction between two modeled components [Pas04].
Transaction-level modeling allows designers to explore and validate implementations at a higher level of abstraction, speeding up simulation and enabling earlier architectural decisions. Alternative implementations can also be explored at early stages of development to prevent costly redesign during later developmental stages.

2.3.2 SystemC

SystemC is a standard design and verification language that can be used from concept to implementation for both hardware and software [Ope02, GLMS03]. SystemC is a C++ library aimed at system-level modeling and has the benefits of C++, an object-oriented language that allows for data encapsulation and generic programming techniques. One of the main goals of SystemC is to enable system-level modeling. Because there are a wide range of computation and communication models, as well as different abstraction levels and methodologies used in system design, SystemC uses a layered approach that provides the necessary flexibility. Software support for SystemC includes an event-driven simulation kernel that handles events and switches between processes without detailed knowledge of the event or process in question. Physical structural elements are represented by SystemC modules and ports, while communication is represented by interfaces and channels. The simulation kernel and these elements form the foundation for implementing models in SystemC. Using SystemC’s data-types and elementary channels, designers can model many systems in SystemC. Besides these core elements, more detailed computation and communication models, libraries, guidelines, and methodologies can be added by the designer on top of the standard SystemC libraries in order to build more complicated system models.

Individual system components can easily be represented as modules in SystemC. A
chapter 2. background

Hierarchical design similar to other hardware definition languages can also be represented in SystemC with ease. Ports define the access window to other components outside of a given module, unknown to that module itself. Channels represent the actual communication between modules without internal knowledge of the modules they are connecting. Communication between modules is through channels that connect modules through their ports and defined interfaces. This black-box approach allows for flexibility as modules can be interchanged as long as the external ports and the type of communication are the same. Hence, SystemC is ideal for the exploration of different architectures.

Simple communication channels are part of the SystemC standard. Typical communication protocols, such as first-in first-out (FIFO), are defined for standard data types. SystemC also allows designers to design their own channels. Designers can create their own structures to pass data between modules, and they can even create their own arbitration scheme. Simple signals are also part of the SystemC standard. These signals are used to communicate within a module and they are also primitive channels. SystemC channels and signals support the request-update method of access. This approach means that writes to these objects do not take effect until the simulation timeline is advanced. Multiple signals are written to sequentially, but their effect is not observed by the model until simulated time progresses. In this manner, SystemC simulates parallel behavior by synchronizing signals and data with simulated time.

The modular aspect of SystemC provides flexibility for the designer, which is important in transaction-level modeling. The transactions between modules are created by processes within the modules or channels in the SystemC model. The functionality of each SystemC module is defined by processes within the module. These processes can be triggered by an
event, such as a clock edge, and the processes are synchronized with the simulated timeline. For example, a pipeline stage performs the same process every clock cycle. The process is defined by an infinite loop that waits to be triggered by an event. Processes are simulated as SystemC threads, and when an event is triggered, all processes sensitive to that event are executed sequentially. Because signals and channels are not updated until simulation time progresses, it appears that the model is executing in parallel. It is in this manner that SystemC can easily model events that take place in parallel [GLMS03].

2.3.3 Trace-driven and Statistical Simulation

Trace-driven simulation is an existing simulation technique that takes an application’s instruction trace and executes the instructions from the trace file. The technique is accurate, but the approach requires large trace files to be stored [ENSB03]. For multi-threaded processors, multiple representative instruction traces would be required to provide instructions for multiple instructions streams. However, this would require even more storage space and time to produce. Therefore, in order to reduce space requirements, a statistical approach can be used.

A synthetic trace can be generated from an application model in order to drive a simulation. The basic methodology is to obtain an application’s representative execution trace, profile the application based on such a trace, and generate a synthetic trace for subsequent simulation experiments. A synthetic trace requires a fraction of the space required for the original trace file, and it can produce a trace with similar characteristics and behavior as the longer original trace.

Statistical simulation can be taken further by using statistical models to simulate certain
aspects of the processor model. For instance, the detailed functional cache model could be replaced with a statistical cache model [ENSB03]. The response of the statistical cache model is based on probability rather than the simulated contents of the cache and memory over simulated time. However, determining the probabilities for a statistical cache model is not easy for a multiprocessor with the added complexity of cache coherence.

2.4 Summary

This chapter discussed topics relevant to the work of this thesis including processor architecture, software modeling, and simulation. In subsequent chapters, a software model of a multi-core multithreaded processor and an application model will be discussed in detail. The use of the processor model and the application model in order to simulate and explore different hardware configurations is subsequently discussed.
Chapter 3

Model of a Processor

This chapter presents a transaction-level model (TLM) of a multi-core multithreaded processor that was implemented in SystemC. Software modeling and simulation are common techniques used to aid hardware design. Through simulation, different processor architectures can be explored and verified at early stages in the development cycle. Details of the architecture modeled are discussed, followed by the implementation of modules used to model the processor in software. The software model’s implementation is discussed in two sections: the components and the communication. Finally, this chapter concludes with a discussion of the usage of these modules that enable systems with varying processor configurations to be modeled.

3.1 Architecture

The architecture modeled is based on Sun Microsystems’ chip multi-threading (CMT) architecture described in Section 2.2. Figure 3.1 and Figure 3.2 provide details on the multi-
core multithreaded processor architecture that is modeled. It consists of \( n \) independent processor cores connected to a shared L2 cache and an external memory system, as illustrated in Figure 3.1. Each processor core has the ability and the necessary hardware to swap between \( m \) threads. A fine-grained multithreading scheme was implemented for the processor model, as described in Section 2.2.

As shown in Figure 3.2, the processor core contains separate L1 instruction and data caches. The L1 caches gain access to the external bus and the L2 cache by an external core interface. Physically, the cores require additional sets of the programmer-visible registers to maintain per-thread register data and allow threads to be swapped in and out of operation without software overhead for saving and restoring registers. At any point in time, only one set of registers within a processor core is active. The active thread and its corresponding registers operate as a single pipelined processor. Instructions are fetched and each instruc-
Figure 3.2: Block Diagram of Processor Core

tion is processed through the pipeline in subsequent clock cycles. The pipeline consists of fetch, decode, execute, memory and commit stages. Instructions that do not require access to the memory stage are processed from the execute stage directly into the commit stage. This allows work to be performed in the event of a lengthy cache miss. When a memory stage’s memory request results in a cache miss, the active thread is swapped out and its registers are disabled. The new active thread is allowed to begin operations and a different stream of instructions is processed by the processor. Because instructions that do not require memory access bypass the memory stage, the instruction stream can proceed even if the memory stage is stalled. However, when the execute stage is processing a load or store instruction, and the memory stage is currently stalled due to the memory access of another
thread, the active thread is swapped out. When threads reactivate, the required pipeline structures re-execute the swapped out instructions and operations resume for that stream of instructions.

### 3.2 Memory

As described in Section 3.1, each processor contains its own L1 data and instruction caches. The processor cores share a L2 cache and share access to main memory.

The cache can be configured differently based on the following parameters: set associativity, cache capacity, and block size. Besides the configuration, two other important issues need to be addressed: cache replacement and cache coherency. A least-recently-used (LRU) cache replacement policy was chosen. If the cache is full and a new entry is required, the cache entry that has least recently been accessed is removed to make room for the new entry. The cache coherence policy chosen is the MOESI protocol. The protocol was described earlier in Section 2.1.2.

The L1 cache is not only required to service local cache requests, but also external requests due to cache coherence. To reduce the waiting time for snoops, the L1 cache has been implemented with the ability to be accessed twice simultaneously. This dual L1 cache access is implemented in hardware by using two sets of address tags and corresponding comparison logic [HP03]. The standard cache access is a request resulting in a cache hit/miss check and a potential data output. The secondary access performs the cache hit/miss check for snoop requests to provide a rapid response for cache coherence. This secondary cache request is only used to check that status and does not provide data or
change status information. If a snoop hit occurs for the secondary cache access, a primary cache access request is made to either provide data or adjust the cache status information.

3.3 System Implementation

SystemC 2.1 v1 was used to implement the transaction-level model (TLM) of the multi-threaded multi-core processor. A transaction-level model hides low-level detail from the designer. Details such as the exact computation and actual data are not necessary for this level of simulation. This enables designers to reduce the complexity of the model and focus on the communication and the interaction between modeled components. Because of this reduced complexity, TLMs are ideal for early architectural exploration. For a processor, a TLM can focus on memory interaction, which is one of the limiting factors of a processor’s design. The model can hide the details of arithmetic operations or other computation operations that do not play a large role in performance. This simplification allows different designs to be easily explored and verified before further production steps are taken.

The model is implemented by using SystemC modules and interconnecting channels to represent the components and communication within a processor and memory system. The modules contain clock-driven processes that may create other events such as a memory request. SystemC executes all of the events for a given instant of simulation time sequentially, but the state of the model does not change until the simulation time has progressed to the next instant. This gives the effect that the processes are executing in parallel. The synchronization of these processes is handled by SystemC.

The flexibility of SystemC allows the created components and channel modules to be
reused within the design. Modules such as the processor core can be reused to build larger, more complex processor models in order to explore different processor configurations. Details of the processor model, including the hierarchial use of modules, are discussed in depth throughout the following sections.

Overall, the SystemC-based processor model implemented is comprised of 12 source files and 28 header files with an approximate raw line count of 14,000. This includes the multiple header files associated with the different processor configurations and workload scenarios that will be discussed in Chapter 5.

3.4 Component Modeling

The processor contains common building blocks and lends itself to a hierarchial, modular design. The processor can be divided into the processor core and the L2 cache. The processor core can further be divided into the individual pipeline stages, the instruction and data caches, and the memory interface.

Because SystemC data and control signals are not updated until the next simulated time instant, module processes are designated as either positive or negative clock triggered events. This approach enables events within a single clock cycle to be placed into two halves of the cycle, allowing control signals and data to be set during the first half of the clock cycle, and ensuring that the proper value can be accessed during the second half of the clock cycle. These phases may not reflect physical events of the processor but are used to synchronize the control signals within a clock cycle. Signals that are used to make decisions during the same clock cycle are set during the first phase of the clock. The
decisions are then made during the second phase with valid information. For example, the
decision whether to swap a thread is based on events that occur during the same clock cycle.
If there is a memory stage cache miss, the proper signals must be set during the first half of
the clock cycle, to ensure that the proper value is read in the second half of the clock cycle.

3.4.1 Cache Implementation

3.4.1.1 Cache Model

Because cache configuration is an important factor in determining system performance,
flexibility is a key issue when designing the cache model. The cache can be defined by
the cache capacity, the block size, and the set associativity. These parameters define how
the cache is organized physically, and the number of bits required for the tag content. To
determine if a cache request is a hit or a miss, tag and status information are maintained for
every cache entry. A cache hit occurs when a tag entry in the cache at the corresponding
address index is the same as the tag bits from the current address, and the entry in the cache
is valid.

In a TLM, the actual data contents of the cache are not required for modeling purposes,
but information that leads to the determination of a cache hit or miss is required. Tag and
status information is required for every entry of the cache. As discussed in Section 3.2,
a MOESI cache coherency protocol is used in this model, therefore every cache entry has
associated status information which could be either M, O, E, S, or I.

The cache model is implemented as a class object in SystemC that is configured on
instantiation. The cache is defined by the total capacity, set associativity, and block size.
The configuration defines the number of cache entries, the number of entries assigned to each index, and the number of tag bits. Arrays are created to represent the cache tag and status information, and are indexed using the set number and the index.

In the event that the cache is full, a cache replacement policy is used to determine the outcome. The policy chosen in Section 3.2 is the least-recently-used (LRU) policy. For every index, a history list is created to keep track of the most recently used entries. These lists are updated on every cache access so that they may be used in the event that a cache entry needs to be replaced. The cache entry being replaced may need to be written back, depending on the assigned status information.

The main function of the cache module is to read from and write to the cache. Because the data is not modeled in this cache model, the SystemC object must maintain only the status and tag information of the cache. The result of the read or write, and corresponding resulting reactions, are modeled by the internal memory communication interface that will be discussed in detail in Section 3.5.3. The read function requests information regarding the cache status, and that information is passed back to the calling function. This information includes the MOESI status information and whether the request was a hit. Not only does the function provide relevant cache status information, it also provides information that may be relevant to the write function. If the cache is full and a cache miss has occurred, the read function provides the location of the data that needs to be replaced by searching the most-recently-used cache list. Any necessary actions are performed by the calling function. The write function in the SystemC cache model assumes that the cache is ready to accept a write request and replaces the tag bits, and sets the status information to a given MOESI value, and updates the most recently used cache list. The write function can write at a
given location (supplied by a previous read) within the cache, or the first empty entry for
the corresponding index. A simplified version of the cache model header file in SystemC is
shown in Figure 3.3 with the main entities and functions of the class. The snoop function
in the SystemC cache model performs the same tasks as the read function: it requests cache
status information for the address in question.

```c
#define ADATAS uint32
typedef unsigned long int uint32;
struct cacheresponse
{
    char cachestatus;
    bool cachehit;
    ADATAS wbaddr;
};
class cache {
    cache(int clevel, unsigned int csize, unsigned int sets,
          unsigned int blocksize, int numassoc);
    cacheresponse* read_cache(ADATAS address, int dtype);
    cacheresponse* snoop_cache(ADATAS address, int dtype);
    ADATAS write_cache(ADATAS address, int offset, char info);
    ADATAS *cache_data;
    char *cache_valid;
    list<int> *cache_info;
};
```

Figure 3.3: Cache Model Implementation

### 3.4.2 Pipeline Stages

One of the main parts of a pipelined processor is the collection of pipeline stages. In a
transaction-level model, the interaction between elements is the focus of the model. There-
fore, at this level of abstraction, most pipeline stages are exactly the same. The pipeline
stages receive an instruction from the previous stage, and send the instruction to the next
stage. The basic pipeline stage constructor and processes are shown in Figure 3.4. The
class pipeline_basic : public sc_module
{
    sc_in_clk clock;
    sc_port<pipelineread_if> in_iport;
    sc_port<pipelinewrite_if> out_iport;
    SC_HAS_PROCESS(pipeline_basic)
    pipeline_basic(sc_module_name _name):sc_module(_name)
    {
        SC_THREAD(main_action)
        sensitive << clock.pos();
        SC_THREAD(neg_action);
        sensitive << clock.neg();
    }
};
void pipeline_basic::main_action()
{
    current_instruction=in_iport->read();
}
void pipeline_basic::neg_action()
{
    out_iport->write(current_instruction);
}

Figure 3.4: Implementation for Basic Pipeline Stages

variables in Figure 3.4 that have the prefix ’’sc-’’ are standard signals and entities defined by SystemC. For example, sc_in_clk is a standard signal can be used to synchronize the entire model based on a clock. sc_port is a port that interacts with entities outside the module based on the interface that defines it. The functions, pipeline_basic::main_action and pipeline_basic::neg_action, are processes that are triggered by the positive and negative edges, respectively, of the clock. They represent the actions that the pipeline stage will perform during each half of the clock cycle. The port interfaces, pipelineread_if and pipelinewrite_if, are used to accept an instruction from the preceding pipeline stage and pass on the instruction to the next stage in the pipeline. Further details on the communication between stages are described in Section 3.5.1.
class pipeline_cacheaccess : public sc_module, public data_if
{
    ...
    sc_port<reqtop_if> cache_req;
    ...
    virtual void xfer_done();
    virtual void xfer_done2();
};
void pipeline_cacheaccess::main_action()
{
    current_instruction=in_iport->read();
    ...
    cache_miss = true;
    mem_request *req = new mem_request(current_instruction);
    nblocknowaitreq(req);
    wait(data_received);
    cache_miss = false;
    ...
}
void pipeline_cacheaccess::xfer_done()
{
    notify(data_received);
}
void pipeline_cacheaccess::xfer_done2()
{
    wait();
    notify(data_received);
}
void pipeline_cacheaccess::neg_action()
{
    ...
    while (cache_miss)
    {
        ...
        wait();
    }
    ...
    out_iport->write(current_instruction);
}

Figure 3.5: Cache Access for Pipeline Stages

The fetch and memory pipeline stages require interaction with the memory system. Instructions are read from the cache in the fetch stage, and data is read from or written to the cache in the memory stage. The basic pipeline stage in Figure 3.4 has been
extended to include details for memory requests in Figure 3.5. The cache is accessed during the positive half of the clock (pipeline_cacheaccess:main_action) and when the data is valid, the negative half of the clock (pipeline_cacheaccess:neg_action) continues the stage’s functionality and completes the stage. SystemC events are used to notify the pipeline_cacheaccess:main_action process when the data is valid in the cache. The signal, cache_miss, stays asserted until the positive edge process is notified by the data_received event. While the cache_miss signal is asserted, the negative process remains in a loop until the next negative clock cycle where the cache_miss signal is de-asserted. Once the pipeline_cacheaccess:main_action process has been notified of the memory request’s completion, the process continues and de-asserts the cache_miss signal. During the negative half of the clock cycle, the pipeline_cacheaccess:neg_action process exits the cache_miss loop and the stage completes. xfer_done() and xfer_done2() are functions defining the interface used to notify the pipeline stages by the memory system. In the event of an immediate local L1 cache hit, the function xfer_done() is used to notify the requesting pipeline stage without a delay or penalty. xfer_done2() is used to notify the pipeline stage when the cache becomes valid after an initial local L1 cache miss. A one-cycle penalty occurs to ensure that the data is valid at the correct time in the clock cycle. More details of the interface and internal memory communication details are described in Sections 3.5.2 and 3.5.3.

3.5 Communication Modeling

Interactions between modules in SystemC are defined by interfaces and channels. The interfaces define how a module or channel is accessed. There are three interconnects in the
processor that must be modeled: the interconnect between pipeline stages, the internal core memory interconnect, and the external memory interconnect. Every interconnect maintains a list of requests that are handled by the interconnect’s processes.

### 3.5.1 Pipeline Stage Communication

The interconnect between pipeline stages represents transactions between them, as an instruction moves through the pipeline. Each pipeline stage reads from the interconnect the instruction that it should process.

A custom channel, shown in Figure 3.6, was created to connect pipeline stage modules together, including the connection between the instruction generator and the first pipeline stage. This channel passes an instruction object created by the instruction generator module between stages. The instruction object contains data relevant to the model such as instruction type, instruction and data addresses, and the thread identification. This information is used to determine the activity of each pipeline stage as the instruction passes through the pipeline. When a positive clock event occurs, one of the surrounding pipeline modules reads the instruction from the channel.

![Figure 3.6: Pipeline Interconnect Channel](image)

The interface for communication between pipeline stages is defined in Figure 3.7 and
Figure 3.7: Pipeline channel Interface

An example usage of the interface is shown earlier in Figure 3.4. Due to the multithreading support, pipeline stages may be required to re-execute an instruction when the stage is reactivated. Control signals are used to ensure that the pipeline stages execute the correct instruction. Usually, pipeline stages initiate a pipeline channel read through their in-port, as shown in Figure 3.4. The function brings the current instruction object into the pipeline stage for processing. When the stage is complete, the pipeline stage uses an out-port function, write(), to place the object onto a exit channel, which it holds until the next clock cycle. If the thread is being swapped back in, each pipeline stage uses the readlast() function of the out-port to fetch the instruction object it requires in order to reprocess it.

### 3.5.2 Memory Request and Response Interface

This section outlines the interface used for memory requests and responses for both internal and external memory transactions. The control and use of these interfaces are defined by the channel, and are described for internal and external communication in Sections 3.5.3 and 3.5.5.
A memory request will be described as a memory interaction that does not have valid data available and moves away from the pipeline stage (fetch or memory) that requested it. A memory response is the result of a memory request that has found the requested data and is trying to bring the valid data to the requesting element. A memory response moves towards the pipeline stage that requested it and ensures that the elements along the memory path are available for the data transfer. The longest route of a memory request not involving cache coherence includes the pipeline stage that initially requests it, the L1 cache, the external core interconnect, the shared bus, the L2 cache, and main memory. Valid data may be found at any memory element in the described path thereby shortening the route, and the data is passed on to the requesting element. A memory request moving away from the pipeline stage that requested it will be described as moving down the memory request path, and a memory response returning to the pipeline stage that requested it will be described as moving up the memory path. When cache coherence is involved, snooping requests are cache inquiries due to the snooping of the shared bus. A snooping response is the result of a snoop hit, and the corresponding cache reaction. The snoop response is based on the cache coherence protocol and may require the L1 cache to provide data to another processor. Such snooping responses require an interaction between two local L1 caches through the shared bus. Other snoop responses may only result in local L1 cache activity and do not require access to the shared bus.

The interface between modules and channels for memory requests moving down and up the memory access path is defined in Figure 3.8. Non-blocking and blocking accesses are declared by the interface, but are defined in the module or channel that uses them. Top-level pipeline stages require an additional interface that allows the request to be performed
during the same cycle. This is outlined by the `reqtop_if` interface, and if an initial memory
request from a pipeline stage results in a cache hit, the pipeline stage will be notified during
the same cycle using the `xfer_done()` function declared by the `data_if` interface shown in
Figure [3.9].

```cpp
class req_if:public virtual sc_interface
{ public:
  virtual void nonblockreq(mem_request *creq)=0;
  virtual void blockureq(mem_request *creq)=0;
  virtual void blockdreq(mem_request *creq)=0;
};
class reqtop_if:public virtual sc_interface, public req_if
{ public:
  virtual void nblocknowaitreq(mem_request *creq)=0;
};
```

Figure 3.8: Memory Request Interface

At all points during a memory access, the stages are waiting for the memory response.
Once the data becomes valid, the modules along the memory path must be notified in re-
verse order in order to reserve the data path. For the fetch pipeline stage, this notification is
shown by the `data_if` interface and is defined in Figure [3.9]. The `data_if` interface usage is
shown previously in Figure [3.5]. For other elements along the memory path, as well as the
memory pipeline stage, notification is defined by the `req_if` interface. Using blocking func-
tions, the module can synchronize the memory path from source to destination. Using the
interfaces declared throughout the memory path, the stages can be informed and controlled
throughout the entire memory transaction. The memory request may also involve another
L1 cache located in a different core as a result of snooping. A snoop hit that requires data
to be provided requires the use of both upward and downward interfaces to control and
synchronize the transaction between the L1 caches and the external bus.
class data_if:public virtual sc_interface
{
    public:
    virtual void xfer_done()=0;
    virtual void xfer_done2()=0;
};

void pipeline_fetch::xfer_done() {
    notify(data_received);
}

void pipeline_fetch::xfer_done2() {
    wait();
    notify(data_received);
}

Figure 3.9: Pipeline Stage Memory Access Notification Interface

A memory request object is used to pass the request’s information up and down the memory access route. The relevant information includes the address, the type of memory access, the source of the request, and if the data is now valid.

3.5.3 Internal Core Memory Communication

Internal core communication links the pipeline stages, the L1 cache, and the external core interconnect for memory requests. In the event of a cache miss, the memory request is passed to the external core interconnect, to then be passed to the external bus. The connections are illustrated in Figure 3.10.

The channel between memory-accessing elements within the processor core reacts to memory requests when called upon. Separate channels are used for data and instruction caches. The channel must consider two situations: a request that does not have available data, and a response that has available data.

When pipeline elements have a memory request, an object is placed onto the channel through the module’s interface as previously described in Section 3.5.2. During every cycle,
the request list is checked and a request is performed accordingly. The channel performs the cache access, and depending on the result, it either passes the request to an external core interconnect (cache miss), or it indicates that the data is available to the waiting pipeline stage (cache hit) through the interface shown in Figure 3.9.

Memory request objects can also come from the external core interconnect. These memory request objects are typically responses from requests that have previously missed locally, and the request has been fulfilled by external core sources. Interfaces are used to reserve the L1 cache in order to transfer the data from the external source to the L1 cache. Once the transfer has been completed, the tag and status fields in the cache are set based on the MOESI protocol.

The internal memory channel may also need to perform a snooping request. When a snoop request has been passed on by the external core interconnect onto the internal memory channel’s request list, the channel checks the cache for a snoop hit. This can be done in parallel with an internal request as was discussed in Section 3.4.1.1. In the event
of a snoop hit, the external memory channel is notified of the hit and a snoop response is placed on the internal memory channel’s request list based on the cache coherency protocol.

To prevent possible blocking of the external bus, other local cache requests are ignored until the snoop response has been fulfilled. In conjunction with the external core interconnect, the memory path is reserved between L1 caches and the communication can be simulated.

```cpp
class l1cachectrl : public sc_module, public reqtop_if
{
  public:
    sc_in_clk clock;
    sc_port<data_if, 0> data_ports;
    sc_port<req_if> cachemiss_req;
    l1cachectrl(sc_module_name _name, cache *cacheunit
    {
      SC_THREAD(main_action); sensitive_pos << clock;
      SC_THREAD(enterreq); sensitive << clock.neg();
    }
    void nonblockreq(mem_request *creq);
    void blockdreq(mem_request *creq);
    void nblocknowaitreq(mem_request *creq);
    ...
    list<mem_request*> reqlist;
}
```

Figure 3.11: Internal Memory Communication Channel

The key elements of the internal memory channel are shown in the code excerpt in Figure 3.11. The process `main_action` is the heart of the channel and is triggered in the first half of the clock cycle. The main process selects and performs the memory request. Depending on the type of request and the response of the cache, different events may occur. A request resulting in a cache hit causes the channel to use the `data_if` interface to notify the requesting module that the request has been fulfilled. If the cache access is a miss, the request is passed down by the `req_if` interface to the external core interconnect. When this request has been fulfilled, a response is passed back up to the channel through the
\textit{req.if} interface, and the transfer is simulated and the requesting module is notified of its completion. The various interfaces outlined in Figure 3.8 are defined by the channel and they vary in terms of direction and whether they are blocking or non-blocking functions. The \textit{reqlist} is a list of memory request objects that are waiting to be processed. The \textit{enterreq} process is a negative edge-triggered process that moves the requests collected during the cycle into the main request list.

### 3.5.4 External Core Interconnect

Communication between the external shared bus and the internal core communication channel is controlled by the external core interconnect. The external core interconnect only uses the \textit{req.if} interface to outline memory requests and data notification.

The interconnect is used as an entry and exit point into the internal memory channels. The interconnect pools the instruction and data requests of the internal channels into one list for a less complex access into the external bus. The interconnect limits the number of active requests from a single processor core on the external bus to one, preventing excessive use of the external bus.

As shown in Figure 3.12, the external core interconnect defines the functions outlined by the interface. These interface functions are used to add a request to the interconnect list, pass a request up and down, or synchronize the memory transfer when the request has been fulfilled. The main function selects a request from the list and moves the request onto the external bus. The interconnect prioritizes possible requests by their effect on the overall flow of the system. Snoop requests followed by data requests are given priority over instruction requests because the external bus is held for snoop requests, and pending data
class pipeline_access : public sc_module, public req_if {
    sc_in_clk clock;
    sc_port<req_if, 0> data_ports;
    sc_port<req_if> pipe_req;
    SC_HAS_PROCESS(pipeline_access);
    pipeline_access(sc_module_name _name)
    {
        SC_THREAD(main_action); sensitive << clock.pos();
        SC_THREAD(enterreq); sensitive << clock.neg();
    }
    virtual void nonblockreq(mem_request *creq);
    virtual void blockureq(mem_request *creq);
    virtual void blockdreq(mem_request *creq);
}

Figure 3.12: External Core Communication Interconnect

requests cause threads to be swapped and stall the pipeline. Pending instruction requests may not even be for the active thread, therefore they are given the lowest priority.

3.5.5 External Memory Communication

The external memory communication can be split into two channels that represent the external bus and the main memory channel. The external bus has an arbiter that selects a request from the pending bus requests. Snoop requests are created for the other processor cores that are connected to the bus. If the snoop misses or if data will not be provided by the snoop hit, the bus checks the L2 cache. In the event of an L2 hit, the bus attempts to reserve the L1 cache and the transfer is simulated. If the L2 cache access is a miss, a bus request is passed to the main memory channel’s request list. The main memory channel then simulates the main memory transaction and notifies the bus when it is complete. The bus eventually resumes the transaction as if it were a hit. The external memory communication is shown in Figure 3.13.
class outcachectrl : public sc_module, public req_if {
    sc_in_clk clock;
    sc_port<req_if, 0> data_ports;
    SC_HAS_PROCESS(outcachectrl);
    outcachectrl(sc_module_name _name)
    {
        SC_THREAD(l2control); sensitive <<clock.pos();
        SC_THREAD(l2req2); sensitive <<clock.neg();
        SC_THREAD(arbitrate); sensitive <<clock.pos();
        SC_THREAD(req2); sensitive <<clock.neg();
    }
    virtual void nonblockreq(mem_request *creq);
    virtual void blockureq(mem_request *creq);
    virtual void blockdreq(mem_request *creq);
}

Figure 3.13: External Memory Channel

Figure 3.14: External Memory Communication Interconnect
Figure 3.14 shows key elements of the external memory channel. The *arbitrate* and *req2* threads represent the bus, while the *l2control* and *l2req2* functions represent the main memory. The bus is accessed using the same interfaces outlined in Section 3.5.2, which either add requests to the bus or add requests to whatever the channel is connected to. Because the bus is limited to only one request per processor core, the bus arbitrates in a first-come first-serve basis for memory requests that have not been fulfilled.

### 3.6 Thread Modeling

Multithreading is a key component of the CMT architecture that needs to be explored. Adding multithreading support may be complex at some levels of abstraction. However, at the transaction-level of abstraction, thread management is the primary consideration. The model must be able to execute the active thread, maintain the inactive threads, and swap threads when necessary.

Individually, instructions do not differ from one another. However, a stream of instructions collectively becomes a thread. The thread is represented in the model by the current instructions existing within the processor pipeline. The threads are generated by an instruction generator module that creates the instructions and passes them on to the first stage of the pipeline. The instruction generator bases the instructions on trace data or a statistical model that will be discussed in Chapter 4. The instruction generator module uses the same interface and channel as the pipeline stages to pass the instruction to the first stage of the pipeline.

To model multiple threads, multiple sets of the early pipeline stages are instantiated,
including multiple instruction generator modules for each thread. At any point in time, only one thread may be active as physically there is only one set of pipeline components per core. To implement this, an enable signal (*activethreadid*) was added to the overlapping pipeline elements that were described in Section 3.4.2. In addition, a swap-back signal (*swapback*) was created to indicate when a previously active thread returns to execution. These control signals are maintained every clock cycle.

The model determines whether a thread swap is required by checking for certain situations. The thread swapping algorithm is shown in a flow diagram in Figure 3.15. If the memory stage is currently stalled and the active thread for the early pipeline stages are the same as the thread in the memory stage, then the current thread will be swapped. Because non-memory-access instructions may bypass the memory pipeline stage, the swapped-in thread can be processed by the processor. However, if the instruction in the execute stage is a read or a write and the memory stage is stalled, then the current thread will be swapped out. Using the control signals described previously, the pipeline stages can determine which thread is active and whether a thread has been reactivated.

The algorithm is split into two SystemC processes, a detection process and a notification process. These processes are triggered on the positive and negative clock edges, respectively. The positive process performs the swap check that is described in Figure 3.15 while the negative process writes the control signals so that the swap, if necessary, can occur for the next simulated clock cycle.
3.7 System Model

3.7.1 Core Modeling

In a multi-core processor model, there will be multiple sets of the previously described internal core modules. SystemC’s hierarchy design permits a module to be made up of other SystemC modules. This will be useful at a higher level in the model such as the processor core. Figure 3.16 shows the basic processor core module in terms of SystemC modules and channels. The thread control mechanism described in Section 3.6 is also shown.

Figures 3.17 and 3.18 provide a simplified version of the processor core constructor for two threads. The previously-defined modules and channels are instantiated and connected together. Additional threads are created by duplicating relevant modules and channels con-
connected together as shown in Figure 3.16. The thread swapping algorithm described in Figure 3.15 in Section 3.6 is implemented as a process of this module.

3.7.2 Processor Modeling

Figure 3.19 show the processor model in terms of SystemC modules and channels. The processor is made up of a core module that is shown in more detail in Figure 3.16, the external communication channel that is described in Section 3.5.5 and a L2 cache. Multiple cores with multiple-thread or single-thread capabilities can be added by instantiating and connecting more core modules. Figure 3.20 is an example of a two-core processor and how the channels and modules are connected.
pipeline_struc(sc_module_name _name, int id, stat *sdata)
  : sc_module(_name),elementid(id), statdata(sdata)
{
  swap = false;
  icachebusy=false;
  dcachebusy=false;
  SC_THREAD(pipeline_neg)
    sensitive << pipeclk.neg();
  SC_THREAD(pipeline_pos)
    sensitive << pipeclk.pos();
  dcachehit = 0;
  ch_fetch_0 = new pipeline_ch("ch_fetch_0");
  ch_decode_0 = new pipeline_ch("ch_decode_0");
  ch_execute_0 = new pipeline_ch("ch_execute_0");
  ch_fetch_1 = new pipeline_ch("ch_fetch_1");
  ch_decode_1 = new pipeline_ch("ch_decode_1");
  ch_execute_1 = new pipeline_ch("ch_execute_1");
  ch_memory = new pipeline_ch("ch_memory");
  ch_commit = new pipeline_ch("ch_commit");
  ps_fetch_0 = new pipeline_fetch("ps_fetch_0", 0, elementid,
        &icachebusy,statdata);
  ps_fetch_1 = new pipeline_fetch("ps_fetch_1", 1, elementid,
        &icachebusy,statdata);
  ps_decode_0 = new pipeline_decode("ps_decode_0",0);
  ps_decode_1 = new pipeline_decode("ps_decode_1",1);
  ps_execute_0 = new pipeline_execute("ps_execute_0",0,
        &dcachemiss, &memthreadid);
  ps_execute_1 = new pipeline_execute("ps_execute_1",1,
        &dcachemiss, &memthreadid);
  ps_memory = new pipeline_memory("ps_memory", &dcachemiss, 0, &memthreadid,
        elementid, statdata);
  ps_commit = new pipeline_last("ps_commit", elementid, statdata);
  instruction_source_0 = new instruction_gen("instruction_source_0", elementid, 0);
  instruction_source_1 = new instruction_gen("instruction_source_1", elementid, 1);
  cache *i_l1cache = new cache(1, 65536, 1,64, 2);
  cache *d_l1cache = new cache(1, 65536, 1,64, 2);
  i_l1ctrl = new l1cachectrl("i_l1ctrl", i_l1cache,elementid, 0,&swap,
        statdata, &icachebusy);
  d_l1ctrl = new l1cachectrl("d_l1ctrl", d_l1cache,elementid, 1,
        &dcachemiss, statdata,&dcachebusy);
  mem_access = new pipeline_access("mem_access", elementid, statdata);
}

Figure 3.17: Process Core Constructor For Two Threads - Instantiations

3.8 Summary

This chapter detailed the implementation of a multithreaded multi-core processor model
for software simulation. The processor model was discussed in terms of an architecture,
component modeling, communication modeling, thread modeling, and system modeling.
Figure 3.18: Process Core Constructor for Two Threads - Connections
Chapter 3. Model of a Processor

Figure 3.19: Processor Top Level Model

```
sc_set_time_resolution(1, SC_PS);
sc_set_default_time_unit(1, SC_NS);
sc_clock sysclk("sysclk", 10, SC_NS, 0.5, 10, SC_NS, true);
cache *l2cache = new cache(2, 2097152, 1,256, 8);
pipeline_struc *pipeline_00 =
    new pipeline_struc("core00", 0, statdata);
pipeline_struc *pipeline_01 =
    new pipeline_struc("core01", 1, statdata);
outcachectrl *outcache = new outcachectrl("outcachectrl",
l2cache, statdata);
pipeline_00->pipeclk(sysclk);
pipeline_00->mem_access->pipe_req(*outcache);
pipeline_01->pipeclk(sysclk);
pipeline_01->mem_access->pipe_req(*outcache);
outcache->clock(sysclk);
outcache->data_ports(*pipeline_00->mem_access);
outcache->data_ports(*pipeline_01->mem_access);
```

Figure 3.20: Processor System Implementation For Two Cores
Chapter 4

Model of an Application

Application modeling is an attempt to characterize the behavior of the instruction execution and data accesses of a program, in order to reproduce similar behavior without the complexity of the full application code. In this thesis, an application trace file from actual execution is analyzed in order to model the application behavior based on groups of related addresses and the types of instructions. Such an application model can be used to create a synthetic trace for the processor simulation discussed in Chapter 3.

This chapter briefly discusses the method used to generate the application trace file, and then describes the development of the application model in terms of the instruction types and the addresses. The application model applies the concepts used in instruction trace profiling for statistical simulations in order to reproduce a trace with similar characteristics. Specifically, the instruction and data addresses that are accessed by an application are not entirely random but follow various patterns, and the purpose of the model is to represent such patterns without having to execute the actual application code. Other considerations addressed by the application model such as time variability will also be discussed. Valida-
tion of the application model is also included in this chapter. Finally, this chapter concludes with a discussion on application modeling for multithreaded processors.

4.1 Application Trace

To model an application in this thesis, an instruction trace is first generated by executing the code for a given application to collect the sequence of instructions and addresses. The instruction trace is then analyzed based on the techniques discussed in this chapter in order to characterize the behavior of the application. These techniques are implemented in a tool called Shade from Sun Microsystems. Shade can accept most SPARC binaries, and it collects trace statistics based on an emulated execution of the program [Sun97a, CK94]. As an application is executed, instruction trace records are recorded by Shade for executed and annulled instructions. Shade is preprogrammed to record most information that an analyzer might require, such as instruction addresses and data addresses. Using the collected data, a wide variety of analyzers can be written to perform various tasks. The Shade analyzer steps through the trace, and analysis of each entry is performed. For this thesis, an analyzer was written to produce a trace that included the type of each instruction, the instruction address, and the data address for load/store instructions [Sun97b]. A sample of the analyzer’s output is shown in Figure 4.1. The first column represents the type of instruction: 0 for execute (i.e., non-memory-access instruction), 1 for load, and 2 for store. The 2nd and 3rd columns are the instruction and data addresses, respectively.
Chapter 4. Model of an Application

The application trace can be analyzed, instruction by instruction, to determine the frequency of each instruction type. As described previously, the first column in the trace (Figure 4.1) represents the type of instruction. There are three different types of instruction that are relevant in this discussion: execute (i.e., non-memory-access), read, and write. The frequencies are used to define the transition probabilities between states of the instruction type model. A model representing the instruction type is created as illustrated in Figure 4.2.

Figure 4.2 is a three-state model describing how the instruction type is determined. Probabilities $e$, $r$, and $w$ represent the probability that the instruction will be either an execute, read, or write instruction, and these probabilities are independent of the previous state. The sum of these probabilities is one and movement between states occurs after every instruction.
4.2.1 Trace Analysis For Instruction Type

A program was written in C++ to analyze the trace file based on the description of the instruction type model. The pseudocode in Figure 4.3 collects the statistics used to determine the probabilities in the instruction type model.

The trace file is analyzed line by line, and the types of instructions are counted. The batchsize parameter is used to take into account variations in time and will be described below in Section 4.4.

4.3 Address Generation Model

For the purpose of reflecting application behavior, a two-state Markov Model has previously been used to describe the relationship between consecutive addresses [Sud04]. The
While (!endoftrace) {
    for ( 1 to batchsize)
    {
        infile >> type of instruction
        infile >> instruction address
        analyzeI (type of instruction)
        ...
        if type of instruction is execute
            increment countexecute
        else if type of instruction is read
            increment countread
        else
            increment countwrite
    }
    output(stats)
    clear(stats)
}

Figure 4.3: Trace Analysis Pseudocode for Instruction Type Generation Model

two states defined in this model represent random addresses and sequential addresses. The next address to be generated is defined by the state of the model, and the movements between states are defined by the transition probabilities. If the model is in the sequential state, the next address is sequential to the previous address. If the model is in the random state, the next address is randomly chosen.

Figure 4.4: Two-State Markov Model for Address Generation

This simple two-state model, however, is insufficient for modeling a complex applica-
tion and can be elaborated further. The sequential state discussed in the two-state model does not accurately describe the patterns that relates addresses together. Addresses of an application are not only sequential, i.e., one word apart, but consecutive addresses that are related may jump forward or backward in memory. Consecutive instruction may even access the same memory address. This behavior must be more accurately modeled. Similarly, the random state in this two-state model does not accurately describe unrelated addresses of an application because a completely random and independent memory access is rare. To perform even the simplest of tasks, a sequence of instructions is required in an application, and such a sequence amid other sequences is not a purely random event. To address these issues, the two-state model in [Sud04] is extended in this thesis to more accurately model the behavior of an application. The following subsections describe the extensions and the algorithm developed to address the deficiencies of the previous two-state model.

### 4.3.1 Correlation Between Groups of Sequential Instructions

As stated previously, a random address in an application’s execution is unlikely. The address will have some relationship with another address in the application’s execution. Therefore, the random address state described in the previous two-state model can be re-defined as an access to a different group of related addresses. When analyzing an actual execution trace, address statistics can be maintained for a collection of independent address groups. Addresses that are found to be outside a selected proximity range relative to any of the collected independent groups are considered independent, and a new independent group can be created for any such address.

A number of characteristics can used to describe the relationship between addresses
within a group and across different groups:

- group sequentiality probability,
- group frequency,
- maximum length of sequential group.

The probability of group sequentiality is similar to that of the simple two-state model, but a broader definition of sequentiality is used. Unlike strict sequentiality, group sequentiality allows for the possibility of jumps in the instruction sequence, as well as for gaps in the address space for consecutive data accesses. Group frequency is also important because a non-sequential address requires that a new group be selected. Because an application may not spend a uniform amount of time in each group, the frequency of each group is collected during trace analysis to provide the probability of selecting a specific group. The maximum length of a sequence is collected in order to limit the lengths of highly sequential groups that are not significant in the application. Forcing the model to choose a new group when the limit has been exceeded ensures that unfrequent groups are not overly accessed.

Figure 4.5: Model For Generating Addresses in Terms of Groups
The group address extensions discussed for the application model are described in Figure 4.5. The probability to remain in the same group is a function of the group sequential probability, \( S \), and the maximum sequential length, \( M \), of the group in question. The probability to leave the group is a function of one minus the group sequential probability, \( 1-S \), and the maximum sequential length, \( M \). When a transition occurs between states, a new group is determined independently of the state it left. The group frequency probabilities, \( F \), determine which group will be selected when a new group is required. \( S \), \( M \), and \( F \) are characteristics derived from the analysis of an application trace file. \( S \) is based on the probability that the previous and current address are part of the same group, \( M \) is based on the maximum sequential length for a group, and \( F \) is based on the frequency of the group, and is determined by the number of times an address belonging to that group is accessed in the application trace.

### 4.3.2 Correlation Between Sequential Addresses

As previously described, the addresses of consecutive items in the instruction stream may not be strictly sequential, but may still be related to each other. The sequential characteristic in the two-state model has been extended to allow for the possibility of a non-sequential related address. A spread characteristic is used to determine if there is a correlation between consecutive addresses during trace analysis. This spread characteristic is expressed as a range that is used to determine whether the previous and current addresses are related to each other, based on their proximity to each other. An address outside this range is considered unrelated to the current group in the instruction stream.

Another point of interest is the probability of an instruction sequence looping back to
a previously known location. The probability of an address returning to a specific spot in
the instruction stream is characterized from an analysis of the application file in order to
model the loops of an application, and give direction to the address generation model. By
specifying the starting location of a recurring instruction sequence in the address space, the
model can ensure that loops actually jump back to a previously accessed address space, and
the presence of a cache can therefore be exploited.

![Model Diagram](image)

**Figure 4.6: Model For Generating Addresses in Terms of Spread**

Figure 4.6 describes address generation at the level of the address offset selection that
reflects the spread characteristic or range. Once a group is selected with the model de-
scribed in Figure 4.5, the actual address must be generated. The generated address is based
on the model shown in Figure 4.6, and it is either located at an offset from the previous
address, or it is the starting address for the selected group, or a new group. When the gen-
erated address is based on an offset, the actual distance forward or backward is governed by
the range probability $P_i$ in Figure 4.6. Probability $P_{\text{outside range}}$ represents the small chance
that the address to be generated is not based on any of the addresses in the group history.
A new independent group is generated in this case. The range offset probabilities and the new group probability are independent of the state from which they arrived. The probability that the address to be generated will loop back is given by $L$, and the probability that the address will not loop back is given by $1 - L$. $L$ and $P$ are probabilities determined from an analysis of an application trace file. $L$ is the loop back probability, and is collected by counting the number of times the group’s starting address is accessed. $P$ is the set of range probabilities determined by comparing the previous and current addresses to determine the offset between the two.

### 4.3.3 Address Generation Overview

The relationship between consecutive addresses can be characterized as the relationship between groups of addresses, and the relationship between addresses within a group. Using these relationships, a layered approach is used to generate addresses. First, an address group is selected from a list of sequential address groups using the techniques described in Section 4.3.1. Once the address group is selected, an offset is used to generate the new address, as described in Section 4.3.2. The address generation technique is identical for all three types of addresses that an application may access; instruction, read, and write addresses. The characteristics of the three types are independent of each other and are maintained separately.
4.3.4 Trace Analysis for Address Generation

A program was written in C++ to analyze an application trace file as described in Section 4.1 based on the techniques described in the Section 4.3. The pseudocode in Figure 4.7 collects the statistics used to determine the probabilities used to characterize the application’s behavior. For simplicity, the address spreads are maintained globally while the other characteristics described are maintained independently for each group.

The items in an application trace file are analyzed in order, and the instruction, read, and write addresses are analyzed independently of each other. A history of base addresses is maintained for the three types of addresses. As the application trace file is analyzed, base addresses are updated and new entries are added to the history. When base addresses are updated, the proximity range window is moved for the next instruction that uses that base address. The proximity range probabilities are based on the offset to the previous address. For every address, the history of base addresses is searched to find the closest previous base address. If there is no base address within the proximity range, the address is designated as being independent, and a new base address is added to the history. If the address is inside the proximity range, the address is considered related to the closet base address. The appropriate spread and frequency statistics are incremented. If the same base address was selected as the previous address, the address is determined to be sequential and the appropriate statistics are incremented. The previous base address is updated with the current address, and the analysis continues for the next item in the trace file. In every iteration, the base addresses are checked to see if they are too close to each other. If this is the case, the base addresses’ statistics are merged in order to prevent ambiguity in
While (!endoftrace) {
    for (1 to batchsize) {
        infile >> type of instruction
        infile >> instruction address
        analyzeI (type of instruction)
        target = min(instruction address, ihistory[i].addr)
        target.updatestats(istats)
        if (read or write) {
            infile >> data address
            target = min(data address, (r/w)history[i].addr)
            target.updatestats((r/w)stats)
        }
    }
    output(istats, rstats, wstats)
    clear(istats, rstats, wstats)
}

min(address, history) {
    min_distance = min(address-history[i].addr,)
    where i = 1 to history.size()
    min_entry = i
    if min_distance greater than maxdiff
        new entry in history.
        return (new entry)
    else
        return (min_entry)
}

Updatestats() {
    If previous entry is equal to current entry
        increment sequential
    If current address is equal to start address
        increment restart
    Increment offset[min_distance]
    Increment frequency
    Update current address
}

Figure 4.7: Trace Analysis Pseudocode for Address Generation Model

selecting a base address. An attempt is made to avoid this scenario with the selection of
an appropriate proximity range, which is discussed in Section 4.5. When the base address
history is full, the base address with the minimum frequency of reference is removed in
order to accommodate a new base address. The *batchsize* parameter is used to separate the trace file into smaller sections in order to account for time variation and will be described below in Section 4.4.

### 4.4 Model Variation Over Time

Modeling workloads without considering the possible effects of time variability could lead to incorrect simulation results [AW03]. Time variability refers to the difference in workload characteristics over the course of a single execution for a given application. Such variability can lead to errors in simulations based on a model when an actual application’s workload at a single point in time is not represented by its average workload as reflected in the model.

The variations throughout an application’s execution, however, are not entirely random, but are often repetitive in nature and are defined as phases [SPH+03]. These phases are a set of intervals from an application’s execution with similar characteristics. The phases can represent the different types of behavior that comprise the entire application’s execution. For example, one phase could represent a section of the code that is memory bound. Another phase could represent a computationally intensive segment. Research has shown that these phases can be detected in order to model variations in application behavior [SPH+03]. By determining the frequency of each phase, a model can appropriately reflect the variety of behaviors throughout the execution of an application.

The concepts of time variability and phase behavior are addressed for the model in this thesis by dividing the trace into smaller sections, and analyzing the individual sections rather than the trace in its entirety. These sections can be grouped into clusters of similar
characteristics that represent specific types of behavior. The number of sections assigned to each cluster determines the frequency of that type of behavior. The model can then generate instructions based on different clusters as simulated time progresses, thereby representing the different phases over time.

The application model, which was previously defined as an instruction type and address generation model, can now then be described as multiple sets of instruction type and address generation models. This concept is shown in a high-level representation described in Figure 4.8.

The states shown in Figure 4.8 represent a high-level view of the model, and they include the necessary details to determine the instruction type and address for instructions as previously discussed in Sections 4.2 and 4.3. The transitions between these high-level states are determined by the frequency of the clusters described above and are common for all states. Movement between high-level states occurs periodically to model the variations in workload over time. In between these transitions, instructions are generated based on the characteristics of the state selected. To implement the time variation concept, a batchsize parameter, as shown in Figures 4.3 and 4.7, is used to separate the trace file into multiple
The application model uses a multi-layered approach to represent different aspects of an application’s behavior. The application’s behavior is characterized from high level to low level by the phase, the type of instruction, the relationship between groups of related addresses, and the relationship between addresses within a group. The layered nature of the address generation model is described in Section 4.3.3. The flow of the entire application model is represented in Figure 4.9. The application model periodically selects a cluster representing one of the different phases of a program. The application model generates instructions based on the characteristics of the selected cluster. For every instruction generated, the instruction type, the instruction address, and if required, the data address are generated. The instruction type is generated by the three-state model described in Section 4.2. The addresses are generated by the group and spread models described in Section 4.3.1 and Section 4.3.2, respectively. The model selects the base address (group) first, and then selects the offset for the new address.

Figure 4.9: Application Model Flow
4.5 Parameter Selection

The application model requires that certain parameters be chosen: the range of offsets and the length of each batch segment. These parameters affect the accuracy with which the application model can reproduce a trace with similar execution behavior.

The range of offsets is selected to be larger than the longest loop, so that multiple history entries are not created for the same sequence of instructions. For example, consider an instruction sequence that progresses through 1000 instructions before looping back to the start. If range is too small, the correct history entry will not be detected, and a new entry will be created for the same sequence.

The length of each batch segment is dependent on the trace to be analyzed. If the trace is more linear, a longer batch length can be used. If the trace contains numerous jumps that disrupt linear flow, a shorter batch length is required to accurately model this non-linear characteristic.

4.6 Model Validation

In order to validate the application model discussed in this chapter, the processor model discussed in Chapter 3 was configured with a single-core single-threaded processor. The processor model was simulated using an actual application trace and its corresponding application model in order to compare processor performance and assess how closely the model reflects the characteristics of the actual application.

The first application that was used to validate the model is called FuzzyK and it is part of the Aerie software packaged written by Mike Eisen [GE02]. FuzzyK performs
analysis of genes and proteins by k-means clustering to explore the conditional correlation of yeast gene expression. FuzzyK was used in this thesis because it represents a long-running application. The program loads data and performs numerous iterations of analysis on the data. The repetitive behavior is an important aspect of the program, and must not be overlooked by the model if it is to closely reflect the application behavior. The techniques described in this chapter were used to model this application in order to recreate a trace of similar characteristics.

After a series of experiments, an offset range of 1000 and a batch size of 10,000 were chosen. Using these parameters, it was found that the FuzzyK application had 22 different types of workloads. Some of the different workloads can easily be mapped to typical segments of an application such as loading test data, storing data, and computation. Table 4.1 is a summary of a few of the different clusters that can found in the FuzzyK application trace. The instruction type percentages in Table 4.1 are defined for execute, read, and write instructions. Address usage describes how the addresses within the cluster are weighted. An evenly distributed address usage indicates that the base addresses present in the cluster’s history list were used with approximately uniform probability. A heavily weighted address usage indicates that some of the base addresses were more frequently used than others. The address spread refers to how addresses for consecutive instructions varied within a proximity range window. A linear address spread reflects typical behavior of referencing adjacent memory locations, whereas a non-linear spread reflects jumps to different locations for consecutive instructions in a sequence. The loop descriptor refers to the frequency with which a sequence starting from a given base address returned to the base address or near to the base address.
Table 4.1: Sample of Clusters From FuzzyK Application

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Inst Type Prob</th>
<th>Address Usage</th>
<th>Address Spread</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E   R   W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loading</td>
<td>67  33  0</td>
<td>Evenly distributed</td>
<td>I &amp; R Non-linear</td>
</tr>
<tr>
<td>Access</td>
<td>36  55  9</td>
<td>Heavily weighted for I &amp; W</td>
<td>I &amp; R linear &amp; W loops</td>
</tr>
<tr>
<td>Computation</td>
<td>78  16  6</td>
<td>Heavily weighted</td>
<td>Linear, some loops</td>
</tr>
</tbody>
</table>

The model of the FuzzyK application was validated for a linear and non-linear subsection of the application trace as well as in its entirety. For subsection validation, the processor model was simulated for 50 million cycles using both the application trace and a synthetic trace from a partially complete application model, i.e., complete just for the subsection of interest. The processor model was also simulated for the entire application trace and an equal length of simulated clock cycles was then used for the full application model. Table 4.2 summarizes these results for selected subsections and for the entire application. The total number of instructions processed over the same period of simulated time was used to compare the application model’s behavior with the actual behavior of the application trace. The instructions-processed ratio that is described in Table 4.2 is based on the number of instructions processed using the application model over the number of instructions processed using the instruction trace file. For a linear subsection, the number of instructions processed was similar for both application model and instruction trace file simulations. The application model was able to characterize and reproduce the linearity in
the trace file subsection. For a non-linear subsection, the processor simulation using the application model did not process as many instructions as the instruction trace file. This may be due to some of the unpredictability of address sequences. Similarly, the simulation for the full application model that includes both linear and non-linear subsections did not process as many instructions as the instruction trace file.

Table 4.2: Instructions-Processed Ratio for FuzzyK Application and Model

<table>
<thead>
<tr>
<th>Section</th>
<th>Clock Cycles</th>
<th>Instructions-Processed Ratio (Model/Trace)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Subsection</td>
<td>50M</td>
<td>1.037</td>
</tr>
<tr>
<td>Non-linear Subsection</td>
<td>50M</td>
<td>0.899</td>
</tr>
<tr>
<td>Full Application</td>
<td>3950M</td>
<td>0.908</td>
</tr>
</tbody>
</table>

Figures 4.10 and 4.11 illustrate the address spread probabilities for both the application trace (a, c, and e) and the synthetic trace (b, d, f) for the linear and non-linear subsections, respectively. The spreads of the application model for both of these cases are similar to the original spread of the application trace. The non-linear subsection has potentially a larger number of jumps and may use a wider spread compared to the linear subsection. This difference is one of the distinctions that can be used to separate and combine clusters as discussed in Section 4.4.

Figures 4.12 and 4.13 illustrate the instruction and data addresses for a linear section and non-linear subsection, respectively, for the actual application trace (a and c) and the synthetic trace (b and d). Figure 4.12 shows that the linear subsection’s behavior can be reproduced with reasonable accuracy using the model. The actual linear trace subsection did
Chapter 4. Model of an Application

Figure 4.10: Trace and Model Address Spread Probabilities for Linear FuzzyK Subsection
Figure 4.11: Trace and Model Address Spread Probabilities for Non-linear FuzzyK Sub-section
not involve many distinct data address groups, hence the synthetic trace is focussed around a few distinct groups. For the non-linear subsection, however, the synthetic trace does not appear to be similar to the actual trace, as shown in Figure 4.13. Nonetheless, the variety of independent instruction address groups are maintained in the synthetic trace and may have affected the number of instructions processed during simulation. The data addresses for the actual non-linear subsection are not as linear as they appear. The data address groups are separated by a wide margin, but due to the scale of the plot, they appear grouped together. In the generated synthetic trace, these multiple distinct groups are emphasized as they are more spread out.

Regardless of the location in memory for the sequences of addresses that are generated by the application model, the goal is to produce similar execution behavior which would result in a similar amount of work being performed. In order to achieve this goal, similar cache miss rates would have to be generated. The addresses generated had similar address spread characteristics, as illustrated in Figure 4.10 and Figure 4.11, which would result in similar cache performance based on the principle of spatial and temporal locality.

For the FuzzyK application, in summary, the application model produces a similar trace in terms of the address spread, address flow, and the number of instructions processed over the same period of simulated time. In this case, the parameters selected appear to suit the application well. The proximity range does not result in any collisions between base addresses. The batch size produces many different clusters, but only a few dominant ones. These dominant clusters represent the bulk of the application’s execution, and the infrequent clusters are not heavily weighted in the application model.

A second application was also used to validate the model. The application *djpeg* is a
Figure 4.12: Trace and Model Addresses for Linear FuzzyK Subsection
Figure 4.13: Trace and Model Addresses for Non-Linear FuzzyK Subsection
utility program whose purpose is to decompress a JPEG image and output the image to a BMP image file. This second application was chosen because it performs different types of work than the FuzzyK application. The djpeg application requires the reading of a file, as well as the output to a file, in addition to the relevant computation required to decompress the image data in JPEG format. The same trace generation and type of application modeling as the FuzzyK application were performed on this application, and the corresponding trace file and application model were used to drive separate processor simulations.

The djpeg application was modeled using a batch size of 1000 and an offset range of 1000. The results of the trace and application model simulations are summarized in Table 4.3. The two most frequent clusters (32% and 21%) in the application model were isolated and simulated for comparison. The number of instructions processed over a fixed period of simulated time was similar for the most frequent cluster and the trace file. However, the second most frequent cluster processed 15% less than the actual application trace file.

Figures 4.14 and 4.15 illustrate the instruction and data addresses for the most frequent cluster (32%) and the second most frequent cluster (21%), respectively, for the actual application trace (a and c) and the synthetic trace (b and d). The instruction and data addresses of the actual trace file for the two cluster are repetitive and concentrated around the same address locations as illustrated in Figure 4.14 (a and c) and Figure 4.15 (a and c). This repetitiveness in instruction and data addresses is reflected in the synthetic traces for the most frequent cluster, as shown in Figure 4.14 (b and d). However, as illustrated in Figure 4.15 (d), one of the second most frequent cluster’s data address streams does not repeat. This would lead to an increase in cache misses, and a decrease in the number of instructions processed, which could explain the differences between the actual application and the
application model’s performance in Table 4.3.

Table 4.3: Instructions-Processed Ratio for jpeg Application and Model

<table>
<thead>
<tr>
<th>Section</th>
<th>Clock Cycles</th>
<th>Instructions-Processed Ratio (Model/Trace)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32% Cluster</td>
<td>50M</td>
<td>0.98</td>
</tr>
<tr>
<td>21% Cluster</td>
<td>50M</td>
<td>0.85</td>
</tr>
<tr>
<td>Full Application</td>
<td>550M</td>
<td>0.757</td>
</tr>
</tbody>
</table>

In summary, the application model can adequately reproduce a synthetic trace with similar characteristics for two applications tested. The linearity, repetitiveness, and the number
Figure 4.15: Trace and Model Addresses for Second Most Frequent Cluster
of accessed address regions of the application affect the ease with which the application can be modeled. A balanced choice must be made between the practicality of the parameters and the desired accuracy of the model because, in practice, there is a limit to the proximity range and the number of stored base addresses in the history. The batch size is also an important factor that affects accuracy. Depending on the program, a long batch size may aggregate the behavior of different phases of execution, and obscure certain interesting behaviors that the program may exhibit.

### 4.7 Multithreaded Application Modeling

Throughout this chapter, the application traces were generated for a single-threaded processor. However, a multithreaded application would not produce an identical trace for all of the threads involved in computation. Hence, the motivation exists to extend the application model discussed in this chapter to reproduce multiple threads with appropriate characteristics for different multi-core multithreaded application scenarios. These different scenarios can be addressed by determining the extent to which there is sharing of a common history of base addresses across different threads and processor cores. These scenarios are summarized in Table 4.4. The internal and external address histories refer to the addresses within a processor core and addresses between processor cores, respectively. If the internal address history is shared, threads from the same processor core access the same address regions. Likewise, if the external address history is shared, instructions from different processor cores access the same address regions. The fourth case, an application with external sharing but without internal sharing is not considered, as it does not reflect a typical ap-
plication behavior. Threads that generate their addresses from a common history of base addresses would simulate the scenario that the threads are working towards a common goal and access the same address space. Threads that generate their addresses independently may represent the same application, but they are separate executions, each using a different address space.

Table 4.4: Address Sharing for Multithreaded Application Modeling

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Internal Address History</th>
<th>External Address History</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent</td>
<td>not shared</td>
<td>not shared</td>
</tr>
<tr>
<td>Thread Parallelism</td>
<td>shared</td>
<td>not shared</td>
</tr>
<tr>
<td>Core &amp; Thread Parallelism</td>
<td>shared</td>
<td>shared</td>
</tr>
</tbody>
</table>

4.8 Summary

This chapter detailed the techniques used to model an application in order to reproduce a synthetic trace with similar characteristics. The application model was discussed in terms of instruction types, address generation, and a time varying model. An approach for multi-threaded application modeling was also discussed.
Chapter 5

Simulation and Experimental Results

Simulation is an essential activity in system design to explore and verify architectural designs. Designers generally use software models as an aid in the development process. These models are used to validate the correctness and performance of proposed hardware designs through simulation before they are built. The work discussed in the previous chapters can be used for this purpose. This chapter documents the use of the processor and application models in simulation experiments and discusses the details of the simulation setup, the different types of experiments performed, and the results.

5.1 Simulation Setup

In Chapter[3] a software model in SystemC for hardware simulation was discussed in order to explore different architectural designs and configurations. Parameters such as the number of cores and the number of threads per core play a role in determining the performance of a system. The flexibility of SystemC and its hierarchical design enables these parame-
ters to be changed with ease. Extra cores are, simply, an additional instantiation of a core module and its channel connections, as described in Section 3.7.2.

The memory architecture described in Section 3.2 is implemented as a queuing system with a shared bus. A flexible cache model allows different configurations to be explored. Memory timing also plays a role in the performance of the system. The memory model allows flexibility in setting the latency for memory transfers such as from the L2 cache to the L1 cache, or from main memory to the L2 cache.

Simulation of the SystemC model is trace-driven using either an application’s execution trace file or the application model that was discussed in Chapter 4. Instructions are fed cycle by cycle into the model to be simulated by the processor model. As discussed in Section 3.6, each application thread is represented by a stream of instructions. The thread’s instruction and data addresses can be generated from varying combinations of common and separate base addresses in order to model different cases of multithreading behavior.

5.2 Experiments

There are many measures of overall system performance for a single processor such as throughput, cycles per instruction, and power consumption. These metrics can also be used to characterize the performance of a multithreaded, multi-core architecture. The metrics that are relevant to this level of simulation and architectural exploration revolve around the total amount of work performed by the processor. The effects on system performance are examined by varying processor configurations, memory timings, and type of workload.

For the experiments that are described in this chapter, the processor model is config-
ured with the cache configuration that is summarized in Table 5.1. Simulation results are obtained with the processor model for a variety of thread and core configurations in order to characterize their performance. Configurations consisting of one, two, four, eight, and sixteen processor cores are considered. For each of these configurations, simulations are performed with one, two, four, and eight threads per core. These processor configurations are used to explore the effects of different processor workloads and the effects of main memory latency.

The workload for the simulations is provided by the FuzzyK application model that was discussed in Section 4.6. Because an application model is used instead of an actual application trace to drive the simulations, the execution of the model is limited to a set period of simulated time in order to provide a common basis for comparison. The number of instructions processed by the system over this period of simulated time is used to compare overall system performance for different configurations. To allow for the cache to be filled, the simulation can also be initially executed for a period of time before the statistics are collected. Figure 5.1 is an example of two configurations that can be explored, and the period of interest is where statistics are collected. The period of interest for the experiments in this chapter is five hundred million clock cycles. Experiments were conducted with and without a setup period of one hundred million clock cycles. The setup period, however, did not significantly affect the results of the simulation. The results discussed in this chapter were conducted without the initial setup period.

The work performed over a fixed period of time and the CPI (cycles per instruction) are used to compare the performance between different processor configurations across different scenarios. CPI is calculated by:
In these experiments, the number of clock cycles is fixed at 500,000,000, and the number of instructions processed varies between processor configurations. As the number of instructions processed increases, the CPI decreases, hence a lower CPI and a higher number of instructions processed are better. Regardless of the scenario, there is a theoretical maximum amount of work that can be performed within the fixed number of simulated clock cycles, and correspondingly, there is an ideal CPI. The best performance is achieved if every clock cycle is used to perform useful work. With single-issue processor cores, the maximum amount of work is calculated by the number of cores multiplied by the number
of clock cycles. The number of threads does not play a role in this calculation because threads within a core do not run in parallel, as shown in Figure 5.1. The ideal CPI is calculated using the maximum number of instructions that can be performed and the number of clock cycles. The ideal CPI is $1/n$, where $n$ is the number of cores that are used.

In the first set of simulation experiments, a variety of workloads, based on the multi-threaded application model discussed in Section 4.7, are used to explore performance for various processor configurations. By simulating different workloads on a variety of different processor configurations, the potential benefits of a multi-core multithreaded system can be explored. The results may indicate under which conditions a multi-core multithreaded would be most beneficial, and under which conditions the architecture may not be beneficial. The results and analysis of these experiments with varying core/thread configurations and workloads are discussed in Section 5.3.1.

The workloads differ in how the instruction streams across different threads and processor cores are related. The addresses generated by the application model are either related or independent, depending on the use of a common history of base addresses. Table 5.2 summarizes the scenarios that are explored. The memory latencies for this set of experiments are summarized in Table 5.3. The latencies include the transfer time between source and destination, as well as the time required to reserve and notify the devices. These latencies assume that the memory path is not being utilized when the request is made.

The second set of experiments is related to one of the main issues in processor design, which is the increasing gap between processor and main memory speeds. The effect of this issue can be explored for the modeled processor architecture by varying the main memory latency. These experiments explore the potential effect of additional processor cores
Table 5.2: Address Sharing for Multithreaded Application Modeling

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Internal Address History</th>
<th>External Address History</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent</td>
<td>not shared</td>
<td>not shared</td>
</tr>
<tr>
<td>Thread Parallelism</td>
<td>shared</td>
<td>not shared</td>
</tr>
<tr>
<td>Core &amp; Thread Parallelism</td>
<td>shared</td>
<td>shared</td>
</tr>
</tbody>
</table>

and multithreading support as the gap between processor and memory speeds increases.

For different processor configurations, simulations are performed with different values for the latency between the L2 cache and main memory. The results and analysis of these experiments with varying main memory latencies are discussed in Section 5.3.2.

Table 5.3: Modeled Memory Latency with Overhead

<table>
<thead>
<tr>
<th>Description</th>
<th>Source</th>
<th>Destination</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache miss &amp; L2 cache hit</td>
<td>L2 Cache</td>
<td>L1 Cache</td>
<td>13 Cycles</td>
</tr>
<tr>
<td>L1 &amp; L2 cache miss</td>
<td>Main Mem.</td>
<td>L1 Cache</td>
<td>23 Cycles</td>
</tr>
<tr>
<td>L1 cache miss &amp; external snoop hit</td>
<td>Non-Local L1 Cache</td>
<td>Local L1 Cache</td>
<td>16 Cycles</td>
</tr>
</tbody>
</table>

5.3 Experimental Results and Analysis

This section details the results of the two main types of performance experiments performed: workload and memory latency. The workload experiments involved three different
base-address generating schemes. The results of these schemes will presented separately, followed by a summary of this set of experiments. The experimental results from varying memory latency will then be described.

5.3.1 Workload Experiments

5.3.1.1 Independent Core & Shared Threads Workload

The first set of results is for the workload where each processor core has its own independent set of base addresses that it uses to generate instruction and data addresses. The threads within a processor core, however, generate instruction and data addresses from a shared set of base addresses assigned to that processor core. This workload corresponds to the thread parallelism scenario in Table 5.2.

Figure 5.2(a) and Figure 5.2(b) show the instructions processed and cycles per instruction (CPI) of the experiments for this workload for different core-thread configurations. The maximum number of instructions and the ideal CPI have been included in Figure 5.2(a) and Figure 5.2(b), respectively, for comparison with the simulation results. Figure 5.2(c) is a plot of the average access delay for a data request. Figure 5.2(d) is a plot of the fraction of time that the shared memory bus is utilized. Figure 5.2(e) summarizes the amount and type of activity on the shared memory bus. Cache coherence activity in Figure 5.2(e) refers to the activity on the bus that results from the MOESI protocol, i.e., a snoop hit that results in an L1 cache providing the data response instead of memory, or an L1 cache sending an invalidation request for a write. The data activity refers to basic memory requests involving the L2 cache or the main memory on L2 misses, where there are no data responses from
another L1 cache.

![Diagram](image)

Figure 5.2: Independent Core and Shared Internal Addresses - Processor Performance

In Section 5.2, CPI was described as a metric for system performance, and improvements in system performance are shown by a decrease in CPI. In Figure 5.2(a) and Figure 5.2(b), the overall performance of the system is improving as the number of processor cores...
is increased. However, the relative increase in performance is diminishing with each additional core. A point will be reached where there will be a negative effect on performance. In this case, the 16-core processor reduces performance shown by a slight increase in CPI in Figure 5.2(b). The diminishing returns on system performance can be explained by the increase in external processor core activity and the bottleneck associated with the shared memory bus. With each additional processor core, the number of bus accesses increases within the fixed period of simulated time for the experiments, as shown in Figure 5.2(e). Correspondingly, as depicted in Figure 5.2(d) and Figure 5.2(c), the bus utilization and the data access delay increase with the number of cores. The delay caused by the memory bus being utilized to the point of saturation by memory activity has a negative effect on performance.

The results in Figure 5.2 can also be used to assess the potential benefits of increasing the number of threads per core with the workload that is considered in this section. As show in Figure 5.2(a), the number of instructions processed increases with each additional thread until there are eight threads per core. The instruction and data addresses generated from the internal set of shared base addresses within each core cause an increase in local L1 cache hits with more threads. The benefits of multithreading are more significant with four, eight, or sixteen cores because the performance benefit from switching to another thread on a cache miss is more pronounced with longer memory access delays. Figure 5.2(c) illustrates that for one or two processor cores, the data access delay is not a significant factor. With four or eight cores, however, the data access delay is longer, thereby providing a greater opportunity for the processor to take advantage of the additional thread support. With sixteen processor cores, the shared bus is saturated, therefore the performance of
Chapter 5. Simulation and Experimental Results

Multithreading is reduced. The benefits of multithreading support are further addressed in the memory latency experiments described in Section 5.3.2.

5.3.1.2 Independent Core & Independent Threads Workload

The second set of results is for the workload where each thread within each processor core had its own independent set of base addresses from which to generate instruction and data addresses. This workload corresponds to the independent scenario in Table 5.2. Figure 5.3 presents results for this workload in a manner similar to the corresponding information in Section 5.3.1.1. Experiments were not performed for the sixteen-core processor because performance would only be further reduced, as shown in Figure 5.3(b).

Using a workload in which addresses are independently generated for each thread within a core, additional threads decrease the performance in all core configurations. The total number of instructions processed in Figure 5.3(a) and the resulting CPI in Figure 5.3(b) show the decrease in performance as more threads are used. There is a marked difference between the results in Figure 5.3 and the results shown earlier in Figure 5.2. This difference is due to the increase in bus activity, as show in Figure 5.3(e), and the resulting bus utilization and data access delay in Figure 5.3(d) and Figure 5.3(c), which can be explained by the increase in L1 cache misses because threads within a core do not share addresses. If the threads are independent of each other, then the benefits of the cache are limited due to the reduced spatial and temporal locality between threads. The additional threads may also cause an increase in cache activity due to thrashing. Data loaded in the cache by one thread might be displaced by another thread, only to be accessed in the future by the first thread, thereby causing another cache miss. The negative impact of additional
Figure 5.3: Independent Core and Independent Internal Addresses - Processor Performance
threads also increases further with additional cores, as there is even more activity on the shared bus.

Any performance increase due to additional processor cores in this workload scenario is also subject to diminishing returns. As described in Section 5.2, a lower CPI is better. Figure 5.3(b) illustrates the performance improving with diminishing returns with each additional core. The onset of negative performance due to bus saturation occurs at different processor configurations for this workload. From Figure 5.3(b) and Figure 5.3(d), it appears that the negative effect on CPI is a result of the bus utilization passing a 95% usage threshold. This corresponds to the bus utilization and performance plots in Section 5.3.1.1 for the independent core and shared thread workload scenario. For that workload scenario, the negative effect on CPI also occurs when the bus utilization is above 95%, as shown in Figure 5.2(b) and Figure 5.2(d).

5.3.1.3 Shared Core & Shared Threads Workload

The third set of results is for the workload where all threads across all processor cores share a single set of base addresses from which instruction and data addresses are generated. This workload corresponds to the core and thread parallelism scenario in Table 5.2. Figure 5.4 illustrates the results for this workload in a manner similar to the results in Section 5.3.1.1. Experiments were not performed for the sixteen-core processor because performance would only be further reduced, as shown in Figure 5.4(b).

The results for this scenario are significantly different from the previous two scenarios. The limiting factor on processor performance is also the shared memory bus, however, the shared memory bus becomes saturated much sooner. In Section 5.3.1.1 and Section 5.3.1.2,
Figure 5.4: Shared Core and Shared Internal Addresses - Processor Performance
the bus activity that led to the decrease in performance was due to data and instruction access from the processor cores, as shown in Figure 5.2(e) and Figure 5.3(e). In this scenario, however, the increase in bus activity is due to the increased interaction between the L1 caches of separate processor cores caused by cache coherence. Figure 5.4(e) illustrates the bus activity and the types of access for this scenario. The snoop activity contributes a higher percentage of the total bus activity than the previous scenarios due to sharing of data between processor cores and the need to maintain coherence. Because of this activity, the shared bus becomes fully utilized at simpler configurations, as illustrated in Figure 5.4(d). Because of bus saturation, there is no performance benefit from additional cores, as shown in Figure 5.4(a) and Figure 5.4(b). Figure 5.4(b) and Figure 5.4(d) also corroborate the observation that the CPI is negatively affected when the bus utilization is above 95% as discussed in Section 5.3.1.2.

5.3.1.4 Summary

The performance of the multi-core multithreading architecture using the simulation model that was described in Chapter 3 is highly dependent on the type of workload that the system is attempting to process. As shown in Section 5.3.1, the best performance is obtained when each core has an independent set of base addresses for instruction and data accesses, but threads within each core share these base addresses. The next best scenario is when all threads are independent of each other. The worst performance occurs when all threads across all cores share a common set of base addresses.

In general, processor performance improves with additional processor cores and multithreading support. These improvements are subject to diminishing returns as the shared
memory bus becomes fully utilized. As the shared memory bus becomes increasingly utilized, memory access delay increases and performance improvements are reduced until performance declines. The memory bus is the limiting factor on how beneficial multi-core multithreading approaches can be.

## 5.3.2 Memory Experiments

A final set of experiments was performed to explore the effect of varying main memory latency on the system. For these experiments, the workload scenario used to generate instruction and data addresses involved each core having its own independent set of base addresses, but threads within each core shared these base addresses. This scenario was used because it previously resulted in the best performance of the three workload scenarios explored, as discussed in Section 5.3.1.1. Experiments were only performed for one, two, and four processor core configurations because the onset of performance reduction due to bus saturation is already near for eight and sixteen processor cores, and any increase in the main memory latency would saturate the bus sooner.

Figure 5.5(a) and Figure 5.5(b) illustrate the CPI and bus utilization for different four-core processor configurations with varying main memory latencies. The main memory latency corresponds to the simulated time modeled to transfer data from the main memory to the L2 cache in terms of clock cycles. Similarly, Figure 5.5(c) and Figure 5.5(d) illustrate the CPI and bus utilization for different two-core processor configurations. Finally, Figure 5.5(e) and Figure 5.5(f) illustrate the CPI and bus utilization for different one-core processor configurations.

As stated in Section 5.2, a lower CPI is a reflection of better performance. Longer
Chapter 5. Simulation and Experimental Results

Figure 5.5: Processor Performance with Varying Main Memory Latency
main memory latencies result in larger penalties for L2 cache misses. L2 cache misses occupy the bus for longer periods of time, and increase overall bus utilization. The effect is clearly reflected in the CPI and bus utilization plots for the four-core processor in Figure 5.5(a) and Figure 5.5(b). The lines representing longer latencies have a higher CPI and bus utilization. Because the bus becomes saturated at less complex configurations, the benefits of multi-core multithreading architectures are reduced with longer latencies.

The higher CPI as a result of longer main memory latency is illustrated in the two-core processor configuration with one thread per processor. For this configuration, the multithreading support has a larger effect on system performance than on the four-core processor because the bus is not saturated. Figure 5.5(c) illustrates that an increase in the number of threads per core generally reduces the CPI for the two-core processor. In Figure 5.5(d), the two-core processor’s bus does not approach saturation, whereas the four-core processor’s bus nears saturation for certain configurations, as illustrated in Figure 5.5(b). The benefits of multithreading support are affected by memory latency. Longer memory accesses provide more opportunity to take advantage of multithreading. As illustrated in Figure 5.5(c), the lines representing shorter main memory latency indicate that CPI is not affected by the number of threads per core. The lines representing longer memory latency show more sensitivity to the number of threads per core. Similarly, the one-core processor’s multithreading performance is relatively uniform for shorter main memory latency, whereas the multithreading support has more of an effect on CPI for longer main memory latency, as illustrated in Figure 5.5(e). Bus saturation is not a factor for the one-core processor, as shown in Figure 5.5(f), because the bus utilization does not approach the threshold discussed in Section 5.3.1.2 that causes a negative effect on CPI.
In addition to the effect of main memory latency, the bus must not be saturated in order for multithreading to be most beneficial to the system. Overall performance is also reduced with longer main memory latency. Therefore, depending on main memory latency, the results obtained with multithreading support may not justify the additional hardware complexity.

5.4 Summary

This chapter detailed the simulation setup, experiments, results, and analysis for multicore multithreaded processor simulation. The processor was simulated with a variety of different thread-core configurations under different workloads and main memory latency timings. The results were used to determine the potential benefits of including additional processor cores and hardware for multithreading support.

From experimentation, it was determined that the best performance occurred when each core had its own set of independent base addresses, while threads within each core shared the set of base addresses to generate instruction and data addresses. The worst performance occurred when each thread within each core shared a single set of base addresses to generate instruction and data addresses.
The main focus of this thesis was the investigation of multi-core multithreaded processor architectures. These architectures are becoming increasingly prevalent due to the diminishing returns of single-threaded improvement techniques. Designers have begun to focus on total performance and methods to increase overall throughput by exploiting thread-level parallelism. This approach takes advantage of multi-core multithreaded architectures and has enabled advances in functionality and performance. However, this approach also increases the design and verification complexity. Detailed modeling of such architectures has become increasingly complex and requires significant development time. An alternative approach of modeling at a higher level of abstraction reduces complexity. Higher-level modeling enables designers to focus on key architectural elements and it allows important architectural decisions to be made at earlier stages in the development cycle.

This thesis describes a processor model and an application model designed to evaluate multi-core multithreaded processor architectures. The processor architectures were modeled in order to simulate different processor configurations and to compare their perfor-
The first contribution of this thesis is a model of a multi-core multithreaded processor architecture. The model was designed at the transaction level of abstraction and written in SystemC. The benefits of a model at the transaction level is that it reduces complexity and therefore the time required to implement and simulate the model compared to other more in-depth and cycle-accurate modeling techniques. The hierarchical nature of SystemC allows for the reuse of modules in order to build the processor model. This flexibility allows different processor configurations to be easily generated and their performance explored. Software simulation was performed using different processor configurations to explore the benefits and limitations of the proposed architecture.

The second contribution of this thesis is an application model that generates traces that exhibit characteristics that are similar to a real application. For long-running applications, the original instruction traces are large, and take time to produce and then use in simulation. In addition, the instruction trace is only one example of an application’s possible execution. To extend an application to multiple instruction streams and to reduce the data storage required to hold the instruction traces, an application model was designed as an alternative simulation workload. The application model produces a synthetic trace with similar characteristics to the original instruction trace. The application model attempts to model application characteristics such as the instruction type, the relationship between groups of addresses, the relationship between addresses within a group, and the variation of the application’s behavior over time. By adjusting various parameters, the application model is able to adequately reproduce a trace with similar characteristics. A program called
FuzzyK was used to validate the application model. FuzzyK performs analysis of genes and proteins by k-means clustering to explore the conditional correlation of yeast gene expression. FuzzyK can represent a typical long-running application; it loads data and performs numerous iterations of analysis on the data. The repetitive behavior is an important aspect of the program that the model must accurately reflect. The processor model was used to compare results from simulations using the actual application trace and the synthetic trace produced by the application model. For the same period of simulated time, the number of instructions processed for each simulation differed by less than 10%. More detailed analysis of the synthetic trace revealed its similarities in terms of address sequences and distributions with respect to the actual application trace.

Using the processor and application model discussed in this thesis, multiprocessor systems were simulated using different processor configurations and different workload scenarios in order to evaluate multi-core multithreaded architectures. From experimentation, it was determined that with more cores and more threads, performance in terms of total instructions processed over the same period of simulated time increased with diminishing returns until the shared bus became saturated. As the shared bus became increasingly utilized, the memory access delay increased and the incremental improvement was lessened until performance actually declined. The cause of the bus saturation, and for which processor configuration its onset begins, depend on the workload scenario. The best performance is obtained when each core has an independent set of base addresses for instruction and data accesses, but threads within each core share these base addresses. The next best performance is obtained when each thread on each core uses an independent set of base addresses to generate instruction and data addresses. This scenario caused an excess of
activity on the shared bus, leading the bus to saturate sooner. The worst performance is obtained when all threads and cores shared the same set of base addresses. This scenario leads to significantly more bus activity due to cache coherence, which saturated the bus and reduced performance. The stalling of processor cores due to the servicing of snoop hits for enforcing the cache coherence protocol also reduced the performance of the multiprocessor for this scenario.

6.1 Future Work

The multi-core multithreaded architecture explored was based on Sun Microsystems’ chip multithreading architecture. From the experimental work, it was shown that the shared memory bus was the limiting factor on performance. Future work could focus on alternative multi-core multithreading processor architectures and different bus sharing schemes. These different architectures may lead to different levels of performance, and provide a solution to the bottleneck occurring at the shared bus. Alternative thread management schemes may lead to better multithreading performance. Instead of the round-robin scheme modeled, a more intelligent thread arbitration scheme could be attempted. The round-robin scheme may activate a thread that stalls and the wasted clock cycles could be better spent on work performed by another thread. Based on the status and the type of instructions of the thread, a more intelligent arbitration scheme could select a thread to be activated that could make more efficient use of the multithreading support.

The application model discussed in this thesis focused on single-threaded application behavior and attempted to generate a synthetic trace with similar characteristics for simula-
tion. The synthetic trace produced by the application model was most similar to the actual application for the more linear subsections of an application’s execution. However, some sequences in the synthetic trace resulted in unpredictable behavior. Some of the unpredictable behavior was addressed by modeling some of the repetitiveness of an application’s execution by determining the number of times that the address sequence returned to, or near, the start of the sequence. This allowed sequences in the synthetic trace to return to a specific point in the address space in order to take advantage of the spatial locality within the cache. This modeled loop, however, may not be the only loop in the sequence. Other loops in the sequence are reflected by the application model through the probabilities of the backwards memory jumps in the address distribution. The jumps could occur at any point in time, and are not guided to any specific point in the address space, leading to some unpredictability in the model. Future work could attempt to characterize these multiple points of interest within a sequence and further reduce the randomness in the application model.

The application model attempted to use the single-threaded application behavior to model a multithreaded application by using different combinations of shared address spaces. Future work could attempt to analyze real multiprocessing applications in order to model their behavior. The correlation between instruction and data addresses within a thread, and the correlation between instruction and data addresses between different threads can be analyzed to model a true multiprocessing application.
Bibliography


