AN APPLICATION-DRIVEN APPROACH TO EVALUATION OF A LIGHTWEIGHT MULTITHREADED ARCHITECTURE

A Dissertation

Submitted to the Graduate School of the University of Notre Dame in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Shannon Kay Kuntz

Jay B. Brockman, Director

Graduate Program in Computer Science and Engineering Notre Dame, Indiana

December 2008
AN APPLICATION-DRIVEN APPROACH TO EVALUATION OF A LIGHTWEIGHT MULTITHREADED ARCHITECTURE

Abstract

by

Shannon Kay Kuntz

“Many interesting applications have arisen whose computational demands for scaling and timeliness stress even our current supercomputers” [42, page 67]. These applications arise from a variety of sources including physics, engineering, biology, sociology, and national defense, to name just a few, and many of them have characteristics that make them ill-suited to conventional high performance architectures. As conventional parallel architectures reach the limits of their scalability, revolutionary new architectures are needed to achieve the performance demanded by these increasingly large and complex applications. This dissertation evaluates one such architecture, a lightweight multithreaded architecture, and explores how multithreaded applications can be organized to take advantage of the architecture’s key features.

The lightweight multithreaded architecture is composed of a network of lightweight processors and memories. The lightweight processors utilize large numbers of lightweight threads of execution to tolerate latencies that conventional architectures cannot avoid. When a given thread of execution is stalled waiting on a long latency event, such as a memory access, the processor can switch to a different thread rather than sitting idle. If there are enough threads available, the processor remains fully utilized, effectively hiding the latency involved in a memory
access. Direct architectural support for lightweight threads with minimal state and low-overhead, memory-based synchronization enable threads to be started, stopped, suspended, and synchronized in only a few cycles. This enables applications to utilize fine grain parallelism with large numbers of dynamic threads and intense synchronization that would restrict performance on traditional parallel architectures.

This dissertation addresses two main questions:

1. How should the architecture be configured to be effective with applications?

2. How should the applications be organized to effectively use the architecture?

In order to answer these questions, experiments and analysis focused on understanding how key factors at the architectural and application level interact to affect performance. It uses an organized approach to evaluation that targets the modeling, simulation, and evaluation tools to the issue under investigation, avoiding the need to simulate and build the entire machine. It then integrates these results to propose an architectural configuration and application programming model. This work comprises the first attempt to code and compare real applications using the architecture’s unique architectural features.
To my parents, John and Elizabeth, and my sister, Heather, for always encouraging me to follow my dreams. To my husband, Todd, for supporting me on the long and winding journey to realize them.
## CONTENTS

<table>
<thead>
<tr>
<th>FIGURES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLES</td>
<td>ix</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td></td>
</tr>
</tbody>
</table>

### CHAPTER 1: INTRODUCTION

1.1 Applications: Driving the Development of Novel Architectures | 2
1.2 Design Challenges | 4
1.3 Objectives
   1.3.1 Key Results | 7
1.4 Dissertation Organization | 9

### CHAPTER 2: RELATED WORK

2.1 Parallel Architectures
   2.1.1 Vector processing | 14
   2.1.2 Multithreaded processing | 14
   2.1.3 Dataflow processing | 15
   2.1.4 Multicore/Chip Multiprocessors | 16
   2.1.5 Integrated logic and memory | 17
   2.1.6 Lightweight Multithreaded Architecture | 17
2.2 Parallel Applications
   2.2.1 Parallel Programming Models: Languages and Application Programming Interfaces | 18
   2.2.2 Evaluation of Multithreading for Dynamic, Irregular Applications | 21

### CHAPTER 3: PROPOSED ARCHITECTURE

3.1 Lightweight Multithreading | 24
3.2 Extended Memory Semantics | 27
3.3 System Organization | 30
3.4 Open Questions | 32
CHAPTER 4: ANALYSIS OF LATENCY, BANDWIDTH, AND THREAD MANAGEMENT ISSUES IN THE BASELINE ARCHITECTURE

4.1 Methodology .................................................. 38
  4.1.1 Statistical Workloads .................................. 38
  4.1.2 Analytical Modeling .................................. 39
  4.1.3 Queuing Simulation .................................. 44
    4.1.3.1 System Level .................................. 45
    4.1.3.2 Lightweight Processor .......................... 46
    4.1.3.3 Memory ......................................... 47
    4.1.3.4 Thread Frame Management ....................... 49
    4.1.3.5 Workload Generation ........................... 50
4.2 Evaluation of Blocking vs. Non-blocking Memory References ................. 50
  4.2.1 Analytical Modeling Results .......................... 54
  4.2.2 Queuing Simulation Results ......................... 63
4.3 Evaluation of Thread Frame Eviction Policy ................................... 78
4.4 Conclusions ................................................................ 93

CHAPTER 5: ANALYSIS OF MEMORY MODELS AND APPLICATION PROGRAMMING ISSUES .................................. 97

5.1 Memory Addressing Models and Programming Issues .................. 98
5.2 Execution Models .................................................. 100
  5.2.1 LWP Simulation Environment ........................ 101
  5.2.2 Cray MTA-2 ............................................. 102
5.3 Comparison of Memory Addressing Models .................................. 106
  5.3.1 Matrix Vector Multiply .................................. 107
    5.3.1.1 MTA-2 ........................................... 108
    5.3.1.2 DGAS Simulation ............................... 109
    5.3.1.3 PGAS Simulation ............................... 111
  5.3.2 Image Histogramming .................................. 112
    5.3.2.1 MTA-2 ........................................... 113
    5.3.2.2 DGAS Simulation ............................... 116
    5.3.2.3 PGAS Simulation ............................... 116
  5.3.3 Heat Conduction .......................................... 118
    5.3.3.1 MTA-2 ........................................... 118
    5.3.3.2 DGAS Simulation ............................... 120
    5.3.3.3 PGAS Simulation ............................... 120
  5.3.4 NQueens .................................................. 122
    5.3.4.1 MTA-2 ........................................... 123
    5.3.4.2 DGAS Simulation ............................... 128
    5.3.4.3 PGAS Simulation ............................... 129
5.4 Evaluation of Active Graphs Using FFT Butterfly Networks .......... 130
# Table of Contents

## Chapter 6: Evaluation of a Massively Multithreaded Architecture for Satisfiability Problems

- 6.1 The Boolean Satisfiability (SAT) Problem
- 6.2 SAT Solvers
  - 6.2.1 Sequential Solvers
  - 6.2.2 Parallel Solvers
- 6.3 SAT Benchmarks
- 6.4 A Multithreaded SAT Solver Implementation
- 6.5 Results
  - 6.5.1 MTA-2 Results
  - 6.5.2 LWP Simulation Results
- 6.6 Key System Characteristics to Support the Multithreaded SAT Problem
- 6.7 Conclusions

## Chapter 7: Putting It All Together: Configuring the Architecture and Applications for Performance

- 7.1 Lightweight Multithreaded Architecture
- 7.2 Multithreaded Application Programming and Execution
- 7.3 Performance Advantages
- 7.4 Conclusions

## Chapter 8: Conclusions

- 8.1 Lightweight Multithreaded Architecture
- 8.2 Application Programming and Performance
- 8.3 Future Work

## Bibliography
## FIGURES

3.1 Conceptual System Architecture ........................................ 26
3.2 Node Organization: Lightweight Processing Chip (LPC) and External DRAM .................................................. 31
3.3 Lightweight Processor Organization (LWP) .............................. 33
4.1 Assembly Code for Vector Sum [17, page 2] ............................... 51
4.2 Blocking versus non-blocking loads [17, page 3] ......................... 52
4.3 Architectural Model for Evaluation of Blocking vs. Non-blocking Memory References .................................................. 54
4.4 Total execution time versus network communication latency for blocking vs. non-blocking memory accesses in analytical model . 58
4.5 Total execution time versus network communication latency for varied number of threads ............................................... 59
4.6 Speedup versus number of processors for varied network communication latency ......................................................... 60
4.7 Effects of number of outstanding memory references on speedup . 61
4.8 Total execution time versus memory access latency for blocking vs. non-blocking memory accesses ................................. 66
4.9 Effects on execution time of varying the memory access latency and number of threads. ............................................... 69
4.10 Speedup versus number of processors for varied memory access latency. ............................................................... 71
4.11 Execution time vs. memory access latency for computation intensive instruction mix ...................................................... 72
4.12 Execution time for increasing network communication latency and number of outstanding memory references ......................... 74
4.13 Effects on execution time of varying the network latency and number of threads. ....................................................... 75
4.14 Speedup versus number of processors for varied network latency.

4.15 Issue Rate and eDRAM Queue Depth for 4 eDRAMS per LPC
(Eviction Limit = 0, Memory Intensive Instruction Mix, Instruction
Miss Rate = 5%) ............................................................... 84

4.16 Issue Rate and eDRAM Queue Depth for 256 eDRAMS per LPC
(Eviction Limit = 0, Memory Intensive Instruction Mix, Instruction
Miss Rate = 5%) ............................................................... 85

4.17 Issue Rate and Mean eDRAM Bandwidth for increasing number
of eDRAMS per LPC (Eviction Limit = 0, Memory Intensive Instruction
Mix, Instruction Miss Rate = 5%) ........................................... 86

4.18 Issue Rate for increasing number of eDRAMS per LPC and In-
struction Miss Rate of 50% (Eviction Limit = 0, Memory Intensive
Instruction Mix) ............................................................... 87

4.19 Issue Rate for increasing number of eDRAMS per LPC and Com-
putation Intensive Instruction Mix (Eviction Limit = 0, Instruction
Miss Rate = 5%) ............................................................... 88

4.20 Issue Rate and Mean eDRAM Queue Depth for increasing number
of eDRAMS per LPC (Eviction Limit = 64, Memory Intensive Instruction
Mix, Instruction Miss Rate = 5%) ........................................... 90

4.21 Issue Rate and Mean eDRAM Bandwidth for increasing eviction
limit (4 Processors, 4 eDRAMS, Memory Intensive Instruction Mix,
Instruction Miss Rate = 5%) ................................................ 91

4.22 Issue Rate and Mean eDRAM Bandwidth for increasing eviction
limit (4 Processors, 256 eDRAMS, Memory Intensive Instruction
Mix, Instruction Miss Rate = 5%) ........................................... 92

5.1 Pseudocode for Matrix Vector Multiply .................................. 107

5.2 Matrix Vector Multiply Performance on the MTA-2 ................. 110

5.3 DGAS Simulation Performance for Matrix Vector Multiply ....... 111

5.4 PGAS Distribution for Matrix Vector Multiply ....................... 112

5.5 PGAS Simulation Performance for Matrix Vector Multiply ........ 113

5.6 Histogram Performance for Uniform Distribution of Image Color
on the MTA-2 ................................................................. 114

5.7 Histogram Performance for Single Image Color on the MTA-2 .... 115

5.8 Histogram Performance for DGAS and PGAS Simulations ......... 117

5.9 Workload distribution for histogram in PGAS Model ............... 117

5.10 Heat Conduction Performance on the MTA-2 ....................... 119
### TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>EMS STATES</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>EXPERIMENTS ROADMAP</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>ANALYTICAL MODEL PARAMETERS</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>DEFAULT ANALYTICAL MODEL PARAMETER VALUES FOR EVALUATION OF NON-BLOCKING MEMORY REFERENCES</td>
<td>56</td>
</tr>
<tr>
<td>4.4</td>
<td>DEFAULT QUEUING SIMULATION PARAMETER VALUES FOR EVALUATION OF NON-BLOCKING MEMORY REFERENCES</td>
<td>64</td>
</tr>
<tr>
<td>4.5</td>
<td>PROCESSOR AND MEMORY PERFORMANCE CHARACTERISTICS FOR VARIED MEMORY ACCESS LATENCY ($P = 16, b = 2$)</td>
<td>67</td>
</tr>
<tr>
<td>4.6</td>
<td>MEAN QUEUE POPULATION FOR VARIED NETWORK DELAY ($P = 16, M = 1024, b = 2$)</td>
<td>74</td>
</tr>
<tr>
<td>4.7</td>
<td>THREAD FRAME EVICTION POLICY PARAMETERS AND ASSOCIATED SWEEP RANGES</td>
<td>81</td>
</tr>
<tr>
<td>5.1</td>
<td>FFT GRAPH CHARACTERISTICS</td>
<td>138</td>
</tr>
<tr>
<td>5.2</td>
<td>EXECUTION TIME FOR FFT RADIX-2 GRAPHS</td>
<td>140</td>
</tr>
<tr>
<td>6.1</td>
<td>EXAMPLE APPLICATION OF DP TO SAT PROBLEMS</td>
<td>157</td>
</tr>
<tr>
<td>6.2</td>
<td>SAT BENCHMARK CHARACTERISTICS</td>
<td>166</td>
</tr>
<tr>
<td>6.3</td>
<td>SPEEDUP FOR SATISFIABLE BENCHMARKS ON MTA-2</td>
<td>169</td>
</tr>
<tr>
<td>6.4</td>
<td>SPEEDUP FOR UNSATISFIABLE BENCHMARKS ON MTA-2</td>
<td>173</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

First and foremost I would like thank my advisor, Jay Brockman. Without his wisdom, gentle guidance, and unwavering support this dissertation would never have been possible.

I would also like to thank my committee members Eugene Henry, Peter Kogge, and Douglas Thain. The time they took to read my dissertation and provide feedback is greatly appreciated.

This work was funded by Cray Computing under the DARPA HPCS Program.
“Although Moore’s law and new architectural innovations enable the computational power of supercomputers to grow, there is no foreseeable end to the need for ever larger and more powerful systems.” [42, page 67]. Many applications in the supercomputing domain consist of models and simulations that are almost insatiable in their demand for higher performance. These demands are driven by the need to analyze larger or more complex models, increase model resolution and accuracy, and provide results more quickly. As conventional parallel machines reach the limits of their scalability, revolutionary new architectures are needed in order to achieve the performance level demanded by these applications as well as to address an emerging class of applications that do not perform well on conventional architectures. The development of such revolutionary architectures itself, however, introduces a unique set of design challenges including an extremely large design space, the inability to leverage models and tools from previous generations of an architecture in order to explore it, and a lack of applications able to exploit new architectural features. This dissertation evaluates a lightweight multithreaded architecture for massively multithreaded applications. It explores architectural tradeoffs for a range of application characteristics to determine how to configure the architecture to best support applications as well as how to organize applications to best utilize the architectural features.
1.1 Applications: Driving the Development of Novel Architectures

The goal of computing at its most basic level is to solve problems. The desire to solve new problems, solve bigger or more detailed instances of existing problems, or solve existing problems faster or more easily drives the continued improvement of both applications and architectures. Performance depends not only on the speed of the architecture but also on the ability of the application to fully utilize architectural features. This is demonstrated by the frequently large gap between a machine’s theoretical peak performance and the actual performance achieved by applications. Recently this concept has been captured by the use of productivity as a metric, which includes not only how fast an application runs on a particular architecture but the effort and time involved in programming the application to do so. Therefore, developing a high productivity system cannot focus solely on peak flops but must also consider the characteristics of the target applications including the architectural features needed to support their performance and the ability to program applications to use those features effectively.

Applications display many different characteristics that influence both the ease of programming and performance for different programming models and architectures. These characteristics include memory access patterns, coarse vs. fine grain parallelism, synchronization behavior, and regular vs. dynamic execution. Furthermore, some of these characteristics may change as the application scales. Many traditional high performance applications exhibit coarse grain parallelism with low synchronization, that are numerically intensive, and that have very regular execution and memory accesses patterns. Conventional architectures have evolved over time to optimize performance for these types of applications. In contrast, emerging applications may exhibit characteristics that make them ill-suited
to conventional architectures including frequent memory accesses with little or no
locality, dynamic execution, fine grain parallelism and/or high synchronization,
driving the need for new architectures.

Supercomputers today range from clusters of commodity components to tightly
coupled, custom processors and memory connected by specialized high-speed net-
works. Regardless of the configuration, the primary difficulty faced by all of these
architectures is getting data to the processors. Over time, processor speeds have
continued to increase at a much faster pace than that of DRAM-based memories
and, because of this fact, the transfer of data between memory and processor has
become a limiting factor in performance. Conventional architectures typically rely
on optimizations to reduce and tolerate system latencies including memory hier-
archies involving multiple levels of cache as well as superscalar processors to take
advantage of instruction level parallelism. Unfortunately, many applications do
not exhibit the high spatial and temporal locality needed to effectively utilize this
memory hierarchy, and the available instruction level parallelism is insufficient to
mask long memory latencies. Furthermore, as system sizes increase, the addi-
tional network latency, synchronization, and memory coherency protocols make
the situation worse – severely limiting the realizable speedup and scalability of
these architectures.

This dissertation evaluates a lightweight multithreaded architecture developed
to address these problems and explores the organization of multithreaded appli-
cations that best utilizes the architectural features. The proposed architecture is
composed of a network of lightweight processors and memories. The lightweight
processors utilize large numbers of lightweight threads of execution to tolerate
latencies that conventional architectures cannot avoid. When a given thread of
execution is stalled waiting on a long latency event, such as a memory access, the processor can switch to a different thread rather than sitting idle. If there are enough threads available, the processor remains fully utilized, effectively hiding the latency involved in a memory access. Direct architectural support for lightweight threads with minimal state and low-overhead, memory-based synchronization enable threads to be started, stopped, suspended, and synchronized in only a few cycles. This allows applications to utilize fine-grain parallelism with large numbers of dynamic threads and intensive synchronization that would restrict performance on traditional parallel architectures.

1.2 Design Challenges

Developing a novel architecture poses a number of significant design challenges, creating an extremely large design space for which models, tools, and applications may not yet exist and many of the accepted rules of thumb do not apply. Furthermore, developing a system that scales to tens, hundreds, or thousands of nodes along with programming complex applications that exhibit multiple kinds of parallelism as they scale greatly increase the complexity involved in coordinating design. Together, these issues demand new approaches to design and performance evaluation.

Computer system design typically relies heavily on detailed architectural simulations at all phases of development. Significant amounts of time and effort are required for simulator development, especially for large and complex systems. This cost is frequently mitigated by leveraging models and tools from the development of previous generations of an architecture or using tool sets such as SimpleScalar \[10\] to model and simulate the system. However for a novel architecture, sys-
tem models and simulators must be developed from scratch. This is particularly
difficult and time consuming early in the design process when the design is in a
constant state of flux. Often, by the time a simulator has been developed and
debugged, the underlying architecture has changed. In addition to the cost of
developing simulators, there is significant effort required to develop benchmarks
to run on those simulators. Without a compiler for the target instruction set,
benchmarks must be written in assembly code, which is extremely time inter-
sive. Conversely, it is difficult to develop a compiler before the instruction set
architecture has been solidified.

Once the simulation tools and benchmarks have been developed, significant
amounts of time may be spent in executing simulations, as a result of both the
relatively slow simulation execution and the large number of simulations necessary
to explore the design space. Detailed architectural simulators can be hundreds
to thousands of times slower than execution on native hardware and this effect
is compounded by the size of the design space under exploration. Freeing the
designer from the restrictions of previous architectures greatly expands the design
options to be explored. It also eliminates many of the rules of thumb that designers
use to guide design. For example, a multithreaded processor may place very
different demands on a cache than does a traditional superscalar processor, making
previous heuristics for cache sizing inapplicable. Finally, simulating a system with
large numbers of nodes further adds to the time and complexity of simulation.

Designing new applications for a novel architecture presents its own complica-
tions. The applications are large and complex and traditional approaches to ex-
tracting performance, such as distributing data to optimize locality, finding coarse
grain parallelism, and limiting synchronization, may no longer apply. Therefore,
new programming models that capitalize on the unique features of both the application and the architecture must be developed. Furthermore, programming languages must be developed to reflect the programming model and provide clear, efficient mechanisms for harnessing unique architectural features.

This work addresses many of these design challenges by using a hierarchical approach to modeling, simulation, and evaluation to address key issues. Because different levels of detail are needed during different phases of design, the choice of modeling, simulation, and evaluation tools is targeted to the issue under exploration. Early in the process, the goal of simulation is to evaluate the baseline architecture making key system level design decisions. Statistical workloads used to drive analytical and queueing-based simulations provide fast development, ease of modification, and reduced simulation time to efficiently explore architectural tradeoffs. Later in the process, the focus changes to the investigation of higher level issues such as programming paradigms, synchronization primitives, and memory models. Execution-based analysis, provided by a detailed simulation environment and execution on a production machine, evaluates programming and performance issues for a particular set of application benchmarks. Targeting the tools to the issue at hand allows evaluation of a number of key architectural and application issues without the need to simulate and build the entire machine.

1.3 Objectives

This dissertation evaluates a lightweight multithreaded architecture for massively parallel applications. It aims to answer two main questions:

1. How should the architecture be configured to be effective with applications?

2. How should the applications be organized to effectively use the architecture?
In order to accomplish this, it seeks to understand how key factors at the system and application level interact to affect performance and then use that information to define an architectural configuration and application programming model. A number of specific architectural and application issues are examined as described below.

**Architectural Issues**

- What are the system bandwidth requirements and what configuration is needed to support them? Specifically, how many processors and eDRAM banks should there be per LPC?

- Should a thread block immediately on a memory access or should non-blocking memory references be supported?

- When should a blocked thread be evicted to memory from the frame buffer?

- Should the architecture support a DGAS or PGAS memory addressing model?

**Application Issues**

- How should applications be structured to best utilize the architecture?

- What programming paradigms and tools are needed to effectively support multithreaded application development and performance? Specifically, how should thread creation and management, data and workload distribution, and synchronization be handled?

**1.3.1 Key Results**

In addressing these issues, this work comprises the first attempt to code and compare real applications using its unique architectural features. As summarized
below, it provides an understanding of system performance trends, demonstrates the effectiveness of key architectural features, and defines a multithreaded application organization to effectively utilize the architecture.

**Architectural Level**

- For a system with unlimited bandwidth, $M_{crit} = L/b$ predicts the number of threads needed to saturate the processor. Increasing the number of threads improves performance up to the point where the processor is saturated.

- Non-blocking memory references reduce the number of threads needed to saturate the processor and increase the latency that a given number of threads can mask.

- Increasing the number of processors improves performance only if there are enough threads available to effectively utilize the additional processors.

- For a system with limited bandwidth, performance is sensitive to the bandwidth requirements. Multithreading alone cannot overcome bandwidth limitations, and so the system must be configured to support the application bandwidth requirements.

- Thread frame eviction traffic further exacerbates bandwidth limitations creating a need for efficient eviction policies. Threads should be allowed to selectively block in the frame buffer to reduce this traffic.

- The architecture is able to support large numbers of fine-grain threads, without the performance hit incurred by conventional architectures, by leveraging architectural support for large numbers of lightweight threads, dynamic thread creation, efficient thread management, and memory-based synchronization.
• A PGAS memory addressing model provides better performance than a DGAS model and requires 1/2 to 1/3 the number of threads to saturate the processor.

**Application Level**

• Multithreaded applications should be organized to

  – exploit fine-grain parallelism with large numbers of small threads
  – use extended memory semantics to support thread synchronization and shared data
  – distribute work and data to take advantage of locality, when available
  – focus on increased parallelism when an application displays little locality or locality that is difficult to extract
  – leverage dynamic multithreading to capture the available parallelism

• In addition to efficient compilers and run-times, the following programming paradigms are needed to support application development and performance

  – explicit programmer control of threads such as that found in DimC’s threaded procedure call
  – high level synchronization mechanisms such as barriers and locks
  – access to low level extended memory semantics
  – parallel control structures for the distribution of work and data

1.4 Dissertation Organization

In this work, a series of experiments is used to evaluate a lightweight multithreaded architecture and multithreaded applications, ranging from the analysis of
tradeoffs in the baseline architecture to the analysis of programming and memory models for a variety of applications. It begins by evaluating a set of architectural tradeoffs for a range of application characteristics to determine the baseline configuration and features best able to support the target applications. Next, a set of application benchmarks are used to explore the memory models and application programming models. Finally, the results of these experiments are synthesized to determine the architectural configuration best able to support the applications and the application programming model best suited to the architecture.

The remainder of this work is organized as follows: Chapter 2 presents the architectures and technologies relevant to the proposed architecture and recent work in application programming and performance evaluation. Chapter 3 describes the proposed architecture and its key characteristics. Chapter 4 provides an analysis of latency, bandwidth, and thread management issues in the baseline architecture. Application programming issues and memory models are explored in Chapter 5. Chapter 6 determines the key system characteristics needed to support a multithreaded Boolean satisfiability solver on the proposed architecture. Chapter 7 combines the experimental results to describe both how the architecture should be configured and how applications should be organized to be most effective. Finally, conclusions and future directions are presented in Chapter 8.
CHAPTER 2

RELATED WORK

The study of parallel architectures and applications encompasses a broad range of research. This chapter provides a brief overview of work relevant to the lightweight multithreaded architecture and the development of multithreaded applications targeted to run on it. Section 2.1 discusses parallel architectures, the technologies they employ, and how they relate to the lightweight multithreaded architecture evaluated in this work. Section 2.2 then discusses parallel programming languages and application programming interfaces (APIs) as well as research related to the development and evaluation of multithreaded applications for dynamic, irregular applications.

2.1 Parallel Architectures

Parallel computers range from clusters of commodity machines to tightly coupled, custom processors and memory components connected by specialized high speed networks. Different classes of parallel computers demonstrate different performance characteristics that impact the way in which they are programmed and the types of applications for which they are best suited.

Most architectures today use complex pipelined, superscalar processors with out-of-order execution to leverage instruction level parallelism (ILP). The proces-
sor evaluates dependencies between instructions in order to issue multiple instructions to multiple functional units in a single clock cycle. Executing more than one instruction per cycle reduces execution time and increases overall processor utilization. The effectiveness of this approach, however, depends greatly on the amount of instruction level parallelism available in a given code. Furthermore, this approach significantly increases both processor size and complexity.

In order to reduce the memory access latency, most conventional architectures also employ memory hierarchies involving multiple levels of cache to leverage the locality of an application’s data. This is very effective for many applications; unfortunately, some applications do not exhibit the high spatial and temporal locality needed to effectively utilize the memory hierarchy. Furthermore, parallel architectures encounter additional network latency and increased complexity related to the communication and coordination of data between processors, and maintaining the coherency of data in multiple caches can further exacerbate this problem.

Two main approaches to communication and coordination are used in parallel computing: shared memory and distributed memory. Shared memory multiprocessors provide a global, shared address space that all processors can access using conventional load/store semantics, simplifying programming and eliminating the need for complex data distribution or messaging. Communication and coordination are provided by reading and writing shared data, and synchronization mechanisms must exist to support this. Systems may exhibit uniform memory access (UMA) latency or non-uniform memory access (NUMA) latency depending on the architectural configuration. Cache coherent NUMA (CC-NUMA) systems incorporate a cache coherency protocol that guarantees the consistency of data across
processors. Shared memory multiprocessors range from small, bus based systems such as the HP NetServer [32] or Sun Enterprise servers [32] to larger systems such as the SGI Altix [2] that employ a scalable interconnect to support hundreds of processors.

In distributed memory architectures, often called message passing architectures, memory is distributed among the processors in the system. Each processor can directly access only its own local memory, and communication and coordination are achieved by passing messages between processors. This makes the system well suited for large, scientific codes that exhibit significant locality. The overhead of message passing, however, is higher than the load/store semantics of shared memory, and distributed memory architectures are often considered more difficult to program because of the complex and explicit distribution and communication needed. Distributed memory architectures typically scale to much larger numbers of processors than possible for shared memory, with some massively parallel machines such as the IBM Blue Gene/L [41] and Cray XT4 [30] composed of hundreds of thousands of processors. In addition, distributed memory machines may also be configured by combining groups of independent computers to form a single system such as a Beowulf cluster [78]. These machines may use message passing or may be configured as clusters of shared memory machines that use a combination of message passing and shared memory programming. The Top 500 today is dominated by Linux clusters which represent approximately 80% of the machines [3]. The top 10 performers, meanwhile, are typically massively parallel architectures such as the IBM Blue Gene/L. The Blue Gene/L held the top spot from November 2004 until June 2008, making it an important architecture for comparison.
Beyond the more conventional shared memory and message passing systems, alternative processing models exist such as vector, multithreaded, and dataflow processing as well as technology to combine multiple processors or processing logic and memory on a single chip. The remainder of this section discusses these alternatives approaches in more detail and their relation the lightweight multithreaded architecture explored in this work.

2.1.1 Vector processing

Vector processors such as the CDC Star-100 \cite{32} and Cray-1 \cite{79} were developed to support massive vector and matrix calculations by using multiple pipelined functional units to operate on vector elements concurrently. Vector processing is effective for large scientific applications dominated by vector and matrix calculations but requires applications to be cast in these terms. Although vector processors dominated high performance scientific computing for many years, they have slowly been replaced by more general purpose processors, many of which now incorporate some vector functionality in the form of SIMD extensions to take advantage of data level parallelism \cite{69, 77, 89, 91}.

2.1.2 Multithreaded processing

In multithreading, applications are divided into streams of instructions, called threads, that be executed in parallel. Multithreaded processors take advantage of this thread level parallelism (TLP) by issuing instructions from multiple threads within a single processor. This approach improves processor utilization by allowing the processor to issue an instruction from a different thread rather than stall when the current thread does not have an instruction available for execution. Mul-
Tithreading may take a variety of forms depending on when the processor switches between threads and how many threads may have instructions executing in the processor concurrently. The earliest multithreaded processor, the Denelcor HEP [85], and its successors, the Tera/Cray MTA [5, 84] and Cray XMT [28], use interleaved multithreading where an instruction is issued from a different thread at each cycle. In contrast, the MIT Alewife SPARCLE processor [4] and Sun Niagara [53] employ block multithreading where the processor switches to another thread on a long-latency event such as a cache miss or failed synchronization. More recently, simultaneous multithreading has gained popularity with IBM’s Hyperthreading Technology (HTT) [54] and the IBM Power5 [25]. Simultaneous multithreading extends wide-issue superscalar processing by allowing multiple instructions to be issued from multiple threads simultaneously each cycle.

2.1.3 Dataflow processing

In dataflow processing, execution is driven by the availability of data, in contrast to traditional processing in which the flow of execution is controlled through the sequencing of instructions. A dataflow program is represented as a graph of data dependencies, exposing the parallelism inherent in the application [92]. Dataflow architectures, such as the MIT Tagged Token Dataflow Architecture (TTDA) [8], Manchester Dataflow Machine [43], and Monsoon [75], consist of hardware that matches data tokens then fetches and executes the corresponding instruction while the P-Risc [71] architecture presents a hybrid of dataflow and traditional control flow models. In spite of the inherent parallelism of dataflow processing, it never gained widespread acceptance due to its significant differences from traditional von Neumann computing and the unavailability of hardware and
software to support it. The dataflow concepts did, however, evolve into the current instruction level parallelism (ILP) found in today’s superscalar, out-of-order execution processors.

2.1.4 Multicore/Chip Multiprocessors

Multicore, also known as chip multiprocessing (CMP), refers to an architectures in which multiple processing cores are placed on a single chip. As technology allows more transistors per chip, performance improvements have historically been realized by creating more complex processors to capture instruction level parallelism. This approach is limited, however, both by the availability of instruction level parallelism in applications and the problems posed by increasing power consumption. Multicore presents an alternative to this approach that focuses on using available transistors to place multiple simple processing cores on a single chip to leverage thread level parallelism. This takes advantage of shorter distances and faster bus speeds to provide better core-to-core communication, lower overhead, and reduced power consumption as compared to a traditional multiprocessor. Dual- and quad-core general purpose processors such as the Sun Niagara [53], AMD Opteron [56], IBM Power5, and Intel Xeon [80], are widespread with 8, 16, and higher numbers of cores becoming more common. Furthermore, router chips and graphics chips today employ upwards of 200 cores. Multicore performance is limited primarily by the ability of software to use it. As multicore processors become more widespread, however, significant research into this area is quickly developing, with Intel and Microsoft currently spending millions of dollars to research this issue.
2.1.5 Integrated logic and memory

Historically, memory and logic chips were manufactured separately because of significant differences in the process. Technological advances in merged logic and memory, however, have made it possible to combine larger amounts of memory and logic on a single chip. In its more common form, this is realized as embedded DRAM [46] which increases the amount of high bandwidth, low latency memory available to the processors. Alternatively, logic can be placed onto a memory chip, transforming the way in which memory works and making it an active participant in computation as envisioned by approaches such as processing-in-memory (PIM) [18] and intelligent RAM (IRAM) [76]. Alternatively, smart memory uses programmable PLAs within a multicore chip to create a reconfigurable architecture [60].

2.1.6 Lightweight Multithreaded Architecture

The lightweight multithreaded architecture explored in this work combines a number of these approaches to develop a novel architecture targeted to applications with dynamic execution and irregular memory access. The system assumes a multicore model with multiple lightweight multithreaded processors and embedded DRAM on each chip. The hardware support for lightweight multithreading found in these processors evolves from the early research in dataflow and multithreading. In particular, the use of frames in memory and split-phase memory operations are based on ideas developed for machines such as the P-RISC and Threaded Abstract Machine (TAM) [31]. The extended memory semantics and hardware synchronization in the frame evolved from the presence bits in the Denelcor HEP and its successors as well as Monsoon’s I-structures [9]. Although not
explicitly modeled in this work, it is also likely that the lightweight multithreaded processors explored in this work could include some type of processing-in-memory and SIMD capabilities as explored in [18]. Further details of the architecture are presented in the next chapter.

2.2 Parallel Applications

Parallel application performance depends on both the parallelism inherent in an application and the ability to extract that parallelism in a way that takes advantage of the system’s architectural features. This section discusses programming languages and application programming interfaces (APIs) for parallel applications as well the development and evaluation of fine-grain, multithreaded applications for dynamic, irregular problems.

2.2.1 Parallel Programming Models: Languages and Application Programming Interfaces

A number of different parallel programming languages and application programming interfaces (APIs) have been defined to support parallel computing on different architectures. The dominant parallel programming model in high performance computing is currently Fortran or C with MPI for parallelism. Fortran and, more recently, C dominate scientific computing, leading to a large body of legacy code. They do not, however, provide many of the higher level abstractions now considered important for software engineering. MPI [66], the message passing interface, is an API standard that provides a set of library routines to support message passing. It provides performance, scalability, and portability with widespread support on a variety of architectures. Developing or porting appli-
cations, however, can be time-consuming and difficult. Programming with MPI requires complex partitioning of data structures and message coordination that often requires significant restructuring of the original sequential program. Although MPI is used on shared memory machines, it is does not have any concept of shared memory and therefore cannot efficiently utilize the architecture’s features.

OpenMP is gaining popularity as a scalable, portable solution for shared memory machines. OpenMP [33] is an application programming interface that supports a shared memory programming model for Fortran, C, and C++ via a combination of compiler directives, library calls, and environment variables. OpenMP implements a multithreaded execution model where the number of threads can change during execution and execution can be controlled through explicit synchronization. It differentiates between private and shared data and can distribute data across parallel loops but provides no real concept of local/remote memory. Shared memory programming is simpler than message passing, and OpenMP can be used to incrementally transform a serial program into a parallel program without extensive restructuring. OpenMP provides performance, scalability, and portability for shared memory machines but still finds itself in the minority as compared to MPI.

POSIX threads, or Pthreads, [36] represents another standardized API for shared memory programming that defines a set of library routines for multithreaded programming. Pthreads is not as well suited to high performance computing as OpenMP for a number of reasons. The thread mechanisms are very low-level and they focus on task parallelism rather than data parallelism, making programming difficult and limiting scalability. The overhead involved in thread scheduling makes it inefficient for fine-grain parallelism. Finally, it was defined
specifically for C and has little Fortran support.

There are a number of parallel languages for shared memory programming. Some are architecture specific, which makes it difficult to port applications between architectures, while others are supported across a variety of platforms. Co-array Fortran (CAF) [72], Unified Parallel C (UPC) [37], and Titanium [93] are all part of a class of languages that support a partitioned global address space (PGAS) and are supported on a variety of platforms. These three languages all use a SPMD model where the number of threads is statically defined at run time and have explicit synchronization mechanisms to control execution. They provide a global view of memory while differentiating between local and remote memory to improve performance. Although they can provide good performance and scalability for many applications, they do not have the widespread support of OpenMP. As a part of the DARPA High Productivity Computing Systems (HPCS) Program’s [1] effort to develop a general-purpose parallel programming language, three new parallel programming languages are under development: Sun’s Fortress [6], Cray’s Chapel [21], and IBM’s X10 [22]. These languages all support a global view of memory with some concept of locality, similar to the PGAS languages discussed above, as well as dynamic numbers of threads/processes and other advanced features. Alternatively, CUDA [73] is an emerging multithreaded language based in C that was originally targeted by NVIDIA to their graphics processors (GPUs). It is becoming increasingly popular, however, as a more general purpose parallel programming language.
2.2.2 Evaluation of Multithreading for Dynamic, Irregular Applications

Recently, interest has grown in the development and performance of fine-grain, multithreaded applications for dynamic and irregular problems. Dynamic problems require complex load balancing and are difficult to scale due to workloads that vary in size during execution. Irregular problems are characterized by irregular data access patterns that do not exhibit the locality needed for cache-based architectures to perform well. Multithreaded architectures exhibit many characteristics that make them well suited to these types of problems.

Early work on the performance of a Pthreads implementation on an SMP is explored in [68] for a set of dynamic and irregular benchmarks: dense and sparse matrix multiplies, two N-body codes, a data classifier, a volume rendering benchmark, and a high performance FFT package. This work finds the existing Pthreads scheduler unable to efficiently support the benchmarks and discusses a set of modifications to the scheduler to improve space and time performance for fine-grain multithreading as required by these benchmarks.

The performance of a dynamic mesh application is explored in [74] for three different paradigms: an MPI message-passing implementation on the Cray T3E and the SGI Origin2000, a shared-memory CC-NUMA implementation on the Origin2000, and a multithreaded version on the Tera MTA. This work examines differences in implementation and performance for the three paradigms. It concludes that the multithreaded version provides the best performance, is simple to program, and provides a very scalable solution.

In [70], the performance of two data-intensive, irregular applications are explored for both the Cray MTA-2 and Sun Niagara platforms. The two applications, a power state estimation and anomaly detection for categorical data, are imple-
mented in Fortran for the MTA-2 and OpenMP for the Sun Niagara with the implementations on both platforms showing significant speedups. Programming for the Niagara was more difficult but the architecture was able to leverage data locality, requiring approximately 25X fewer threads than the MTA-2.

Graph abstractions are found extensively in many different types of applications and exhibit concurrent accesses to shared data structures with little or no locality, limiting performance on many parallel architectures. A multithreaded implementation of a distributed shortest path algorithm for the EARTH-MANNA architecture is presented in [90]. This work highlights the effectiveness of multithreading for automatically balancing the workload during execution and tolerating the long latencies involved in communicating between vertices located in different processors. Results show that the architecture achieves high utilization and almost linear speedup for reasonably large graphs. In [12], Bader, et al explore the architectural requirements for efficient execution of list ranking and connected component graph algorithms on both the Sun Enterprise SMP and multithreaded Cray MTA-2. They discuss differences between the programming models and performance on the two platforms and find that the MTA-2 is simpler to program and demonstrates higher utilization than the Sun Enterprise. These results are then extended in [11] with the development of fast multithreaded implementations of breadth-first search, st-connectivity and shortest paths for unweighted graphs for the MTA-2.
CHAPTER 3

PROPOSED ARCHITECTURE

This chapter sets the stage for the remainder of the work by introducing the generalized architecture under evaluation, highlighting its unique characteristics, and enumerating the main questions to be addressed. The proposed architecture exhibits the following key features:

- lightweight processors with support for large numbers of lightweight threads with minimal state
- representation of thread state as data frames in memory for efficient thread management
- extended memory semantics to support low-cost thread management and memory-based synchronization.

The lightweight multithreaded architecture was developed at Notre Dame through a sequence of projects including DIVA, HTMT, and, most recently, Cascade. Early work evolved from IBM’s Execube [52] combined with the idea that the problem of memory latency could be addressed by moving processors closer to the memory. The Data Intensive Architecture (DIVA) project [35] focused on improving the performance of a traditional workstation by placing processing-in-memory (PIM) chips in the memory hierarchy. This created “smart memories” that could perform selected data-intensive operations directly. The Hybrid
Technology Multithreaded (HTMT) project \cite{88} focused on the development of a petaflop architecture by combining a number of cutting-edge technologies. PIM chips were again placed in memory but with the aim of staging data to feed its super-fast processors. The lightweight processors described in this work were developed in the context of the Cascade project under DARPA’s High Productivity Computing Systems (HPCS) Program \cite{35}. The Cascade “classic” architecture combines heavy-weight processors for computation-intensive execution with lightweight processors close to the memory for more data-intensive execution. My work in architectural modeling and simulation has contributed to architectural design under all of these projects.

The remainder of this chapter discusses the architecture in more detail. Section 3.1 discusses lightweight multithreading including the use of data frames to represent thread state and the multithreaded execution model. Section 3.2 describes the system’s extended memory semantics and how they are used for synchronization and thread management. The system organization and main system components are then described in Section 3.3. Finally, Section 3.4 discusses the open questions under investigation in the remainder of this work.

3.1 Lightweight Multithreading

The proposed architecture utilizes lightweight processors with support for large numbers of lightweight threads of execution to tolerate latencies that conventional architectures cannot avoid. When a given thread of execution is blocked waiting on a long latency event, such as a memory access, the processor can switch to a different thread rather than sitting idle. The effectiveness of multithreading for a given application varies depending on the weight of the threads and the overhead
involved in thread management and synchronization. Heavyweight threads, such as OpenMP [33] threads or Unix Pthreads [20], are typically long running threads that accumulate a large amount of associated state including call stacks, cache data, and branch histories. Creating, synchronizing, and switching between these threads often requires operating system support, thus making these operations very expensive. This limits the number of threads that can be created and the frequency with which they synchronize, as well as making switching useful only for very long latency events. In contrast, lightweight threads with minimal state and direct hardware support for low-overhead thread management and memory-based synchronization enable the system to support large numbers of threads with frequent switching and synchronization. This makes the proposed architecture particularly efficient for applications with irregular, dynamic workloads and/or intense synchronization that are ill-suited to heavyweight threads.

A lightweight thread consists of a small frame of private variables in memory, typically the size of a single cache line. The thread's state, including program counter and register data, are stored in the frame. During execution, the processor accesses the frame's fields directly, treating them as registers. Unlike registers, however, a frame is logically and physically part of the memory system. The frame’s small size and direct accessibility allow the processor to quickly and easily switch between threads of execution without having to explicitly save and restore the contents of registers, although implicit saves and restores occur if the frames are cached. Furthermore, because threads carry their state in frames, they are not tied to a particular processor or limited in number by the processor’s hardware. This allows the system to support large numbers of threads, limited only by the size of memory available for frame storage.
The system architecture consists of a set of nodes connected by a communication network. Conceptually, each node contains one or more lightweight processors (LWPs) and a portion of global, shared memory. This conceptual system architecture is shown in Figure 3.1 with one LWP per node. A process is broken down into a collection of threads and each node contains a subset of these. Threads can be created and destroyed dynamically during execution and are stored in memory using hardware-managed lists of frames. Thread scheduling is managed primarily through two sets of lists: the ready lists (RL) and the blocked lists (BL), with the number of such lists and their distribution being implementation dependent. When a thread has all of the data it needs in its frame to execute the next instruction, the thread is said to be in the ready state. Otherwise the thread is in the blocked state. At each clock cycle a processor chooses a thread for execution from the list of ready threads. As long as enough ready threads exist to execute an instruction on every cycle, the processor remains fully utilized.

Figure 3.1. Conceptual System Architecture
3.2 Extended Memory Semantics

The system’s extended memory semantics (EMS) were developed during the Cascade project as an extension and generalization of the full/empty bits found in the Cray MTA-2 [27] and are used to support efficient thread management and synchronization. EMS relies on the extended double word or Xdword as the fundamental unit of storage throughout the system. An Xdword consists of a 64-bit double word (dword) along with a 65th bit called the extension bit. The extension bit is used with mode fields within the dword to specify a set of extended memory states for that dword. If the extension bit indicates the memory location is full, then the 64-bit field contains valid data. Otherwise, the field contains the metadata needed for the hardware to process the memory reference. Although there are 13 different states defined (some are used only by the OS kernel), only the 7 shown in Table 3.1 are discussed here.

EMS at the memory supports synchronization by allowing memory operations to execute conditionally based on the state of the memory they are attempting to access. Each memory operation specifies a precondition describing the states the memory location must be in for the operation to execute and a postcondition describing the state of the memory location after the operation. Such operations are atomic with respect to other such operations.

Split-phase memory operations, consisting of a request and reply, combined with extended memory semantics allow threads to block in memory, eliminating the need to stall the processor waiting for the result of a memory request. Each memory operation, whether a load, store, or spawn, is packaged as a parallel communication element, or parcel, and sent to the node that contains the target
### TABLE 3.1

**EMS STATES**

<table>
<thead>
<tr>
<th>State</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR</td>
<td>Pending Reply</td>
<td>An operation is pending which will result in the modification of this dword.</td>
</tr>
<tr>
<td>BPR</td>
<td>Blocked Pending Reply</td>
<td>The same as PR except the thread is blocked from execution until the operation completes.</td>
</tr>
<tr>
<td>F</td>
<td>Full</td>
<td>The dword contains a valid value.</td>
</tr>
<tr>
<td>E</td>
<td>Empty</td>
<td>The dword contains an empty value.</td>
</tr>
<tr>
<td>FVE</td>
<td>Forward Value Leave Empty</td>
<td>A store to this dword should be forwarded to the dword indexed by the metadata and the state of this dword should be changed to E after forwarding.</td>
</tr>
<tr>
<td>FVF</td>
<td>Forward Value Leave Full</td>
<td>The same as FVE except the state should be changed to F after forwarding.</td>
</tr>
<tr>
<td>FRK</td>
<td>EMS Fork</td>
<td>A software handler is invoked when the dword is accessed.</td>
</tr>
</tbody>
</table>
memory address. A separate parcel response is generated by the target memory and sent to the requesting thread’s frame at the return location. However, a thread does not stall the processor waiting for the response to arrive but may continue to execute further instructions or block in memory without consuming processor resources. Extended memory semantics are used to mark the return location as waiting for a memory response and, if necessary, place the thread in the blocked state. When the memory response arrives, the result is written to the target location in the thread frame and, if blocked, the thread’s state is reset to ready. In this way, threads do not tie up system resources while waiting for the result of a memory operation and additional instructions from this or other ready threads can be scheduled to keep the processor busy.

For example, a simple producer/consumer synchronization can be implemented using the store.ef and load.fe instructions. The producer uses store.ef to store the value when the location is empty, leaving the location full. The consumer uses load.fe to load the value when the memory location is full and leave the location empty. If the consumer attempts to load before the producer has produced a new value, the load.fe encounters a target memory location in the empty state and is “queued” on the memory location to wait. In the simplest case, this is done by storing the memory operation’s return address in the target memory location and setting the location’s state to forward value leave empty (FVE). When the producer executes its next store.ef, the value is forwarded to the address found at the memory location and the location is set to empty. In this manner, the common case of single producer/single consumer synchronization can be handled by the EMS controller logic at the memory. For more complex cases where multiple operations must “queue” on a single memory location, the fork
state (FRK) can be used to invoke a software handler or additional hardware level queuing. In any case, the need to reissue the operation, which wastes processor and/or network resources, is avoided.

At the thread level, there are two approaches to handling the wait for the return of a memory access: a thread can either block immediately after issuing a memory reference or continue executing independent instructions, blocking only when it needs a result that is unavailable. When implementing blocking memory accesses, the frame location designated to receive the reply of an access is set immediately to the blocked pending reply (BPR) state and the thread is set to the blocked state making it unavailable for execution. For nonblocking memory accesses, the reply location is set to pending reply (PR) and the thread is allowed to continue executing other independent instructions. If a thread encounters an instruction that depends on a frame location marked PR, the location’s state is changed to BPR and the thread state becomes blocked. Writing a memory access reply to a frame location in either the PR or BPR states sets the location state to full and, for BPR, sets the thread’s state to ready. In this way, extended memory semantics avoids stalling the processor when waiting for the result of a memory access (or synchronization) by allowing threads to continue to execute or block in memory.

3.3 System Organization

At the highest level, the system architecture consists of a set of nodes connected by a network. Each node is composed of a lightweight processing chip (LPC) and external DRAM memory as shown in Figure 3.2. Each LPC contains a network interface, a set of lightweight processors (LWPs), and on-chip memory.
Figure 3.2. Node Organization: Lightweight Processing Chip (LPC) and External DRAM
The LWPs are multithreaded processors with direct architectural support for lightweight threads. As shown in Figure 3.3, an LWP consists of an execution pipeline, thread management logic, frame buffer, and instruction cache. The frame buffer replaces traditional register files, providing fast access to and switching between threads of execution and serves as a cache to main memory for frames. The thread management unit chooses a new thread at each cycle from the ready frames using a simple round robin approach. A thread can have only one instruction in the pipeline at a time. This eliminates the need for complicated hazard detection and forwarding hardware. The thread management unit also implements the eviction policy for the frame buffer, determining when and which threads to evict to memory for replacement.

The memory system is composed of both on-chip memory and external DRAM. The on-chip memory is assumed to be embedded DRAM (eDRAM) that can be configured as memory and/or cache. There is no concept of cache coherency between nodes, however, so remote reads cache memory values at their own peril. A memory controller and EMS provide support for split-phase memory operations, frame list management, and synchronization. Local memory requests go to the on-chip memory first and then, if necessary, to the external DRAMs. Remote memory requests are routed to other nodes through the network interface. The system implements a global, shared memory with the memory distribution dependent on the memory addressing model used.

3.4 Open Questions

The architecture consists of a network of lightweight processors and memories with the processors designed to efficiently support large numbers of dynamic,
Figure 3.3. Lightweight Processor Organization (LWP)
lightweight threads. The threads have minimal state and are represented as data frames in memory, reducing the overhead involved in managing threads. Extended memory semantics support low cost, memory-based synchronization and enable threads to efficiently block in memory when waiting for a memory reference or synchronization, eliminating the need to stall the processor or reissue a memory reference.

A number of open questions about the structure and programming of the proposed system existed at the end of the Cascade project, including:

- What are the system bandwidth requirements and what configuration is needed to support them? Specifically, how many processors and eDRAMs should there be per LPC?

- Should a thread block immediately on a memory access or should nonblocking memory references be supported?

- When should a blocked thread be evicted to memory from the frame buffer?

- Should a DGAS or PGAS memory addressing model be used?

- How should applications be structured to best utilize the architecture?

- What programming paradigms and tools are needed to effectively support multithreaded application development and performance? Specifically, how should thread creation and management, data and workload distribution, and synchronization be handled?

The answer to each of these questions will be explored in more detail in the remainder of the dissertation.
CHAPTER 4

ANALYSIS OF LATENCY, BANDWIDTH, AND THREAD MANAGEMENT ISSUES IN THE BASELINE ARCHITECTURE

This chapter presents an analysis of latency, bandwidth, and thread management issues in the baseline architecture. One of the primary features of the massively multithreaded architecture is the ability to support large numbers of very lightweight threads with low overhead management. Two questions related to thread management are tightly coupled to the configuration of the memory system and are used to frame the experiments:

- Should a thread block immediately on a memory access or should non-blocking memory references be supported?

- When should a blocked thread be evicted to memory from the frame buffer?

To answer these questions, experiments examine the effects of latency on performance and the bandwidth requirements under a number of different architectural and application configurations in order to determine the “speeds and feeds” necessary to support large numbers of threads, identify where bottlenecks are likely to occur, and examine the differences in execution between low, moderate, and large numbers of threads. The results are then analyzed to answer the questions posed as well as to determine the architectural configuration needed to support
the bandwidth requirements, specifically the number of processors and eDRAM banks per LPC.

Table 4.1 shows an experiments roadmap for this chapter summarizing the experiments, their focus, and the modeling methodologies used. The first set of experiments evaluates support for non-blocking memory references for a system of simple nodes, employing both analytic models and queuing-based simulations. The second set of experiments explores the thread frame eviction traffic using only queueing-based simulation. It focuses on the bandwidth requirements for a single node to determine a baseline node configuration. Both the analytic models and simulation are driven by statistical workloads. Statistical workloads enable efficient exploration of performance for a broad range of application characteristics. This type of modeling and analysis is particularly useful for coordinating design and evaluating architectural tradeoffs early in the design process. Analytical models provide initial insights into key performance issues while queuing simulations explore these issues in more detail and examine more complex interactions that the analytical models cannot easily address.

The key conclusions drawn from these experiments are:

- For a system with unlimited bandwidth, $M_{crit} = L/b$ predicts the number of threads needed to saturate the processor.

- Non-Blocking memory references reduce the number of threads needed to saturate the processor and increase the latency that a given number of threads can mask.

- For a system with limited bandwidth, performance is sensitive to the bandwidth requirements. Multithreading alone cannot overcome bandwidth limi-
<table>
<thead>
<tr>
<th>Section</th>
<th>Experiment</th>
<th>Focus</th>
<th>Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Blocking vs. Non-blocking</td>
<td>Interaction among a system of simple nodes</td>
<td>Analytic Modeling</td>
</tr>
<tr>
<td></td>
<td>Memory References</td>
<td></td>
<td>Queuing-based Simulation</td>
</tr>
<tr>
<td>4.3</td>
<td>Thread Frame</td>
<td>Configuration of a single node</td>
<td>Queuing-based Simulation</td>
</tr>
<tr>
<td></td>
<td>Eviction Traffic</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

iterations and so the system must be configured to support the application bandwidth requirements.

- Thread frame eviction traffic further exacerbates bandwidth limitations creating a need for efficient eviction policies. Threads should be allowed to selectively block in the frame buffer to reduce this traffic.

- Increasing the number of processors improves performance only if there are enough threads available to effectively utilize the additional processors.

The remainder of this chapter is organized as follows: Section 4.1 discusses the modeling and simulation methodology. Experimental results and analysis for evaluation of blocking vs. non-blocking memory accesses are presented in Section 4.2. Section 4.3 then presents the results and analysis for evaluation of the thread frame eviction policy. Finally, Section 4.4 presents conclusions.
4.1 Methodology

In the design of parallel computing systems, modeling and simulation are used extensively to direct design decisions. This involves significant time and effort including the development of models and simulation tools, the analysis of benchmarks, and simulation execution and analysis. As the size and complexity of both the systems and the benchmarks increase, the time involved in this process can become prohibitive, especially early in the design process when many details are undefined and changes take place frequently. During this early design phase it is important to be able to rapidly prototype, modify, and execute a high-level model of the system to direct further design exploration by identifying areas of interest and concern. Using statistical workloads to drive analytical models and queuing based simulations is one such approach.

4.1.1 Statistical Workloads

Statistical characterization of the system workload provides a way to reduce the time spent in simulation. In evaluating a given architectural configuration, the aim is to determine the performance over a set of applications and data sets that are expected to run on the system. This typically requires tracing the applications and then running these traces through a detailed simulator. These traces can be millions of instructions long and simulation is typically orders of magnitude slower than execution on the computing system making this approach extremely time consuming. Additionally, large numbers of benchmarks may be needed to cover the range of performance characteristics, thus requiring large numbers of such simulations.

In statistical workload characterization, the application is profiled to determine
the distribution of key parameters that accurately characterize its behavior. These key parameters and their distributions can be used as model parameters in an analytic model or used to define a synthetic trace (a priori or on-the-fly) that drives statistical simulation. The statistical trace is much smaller than a full trace and may be faster to execute because low level details of the architecture have been abstracted away. This approach also allows sweeps of the application parameters to explore the design space. Sweeping eliminates the need to trace large numbers of applications, which may be infeasible due to the time involved or because applications do not exist for the architecture under configuration.

In modeling the multithreaded applications for this work, the focus is placed on the number of concurrent threads in the system and the thread characteristics including instruction mix and memory access characteristics. These characteristics are used as parameters for the analytical model or during simulation to determine the synthetic trace. The distributions for these application parameters are determined through application tracing or hand analysis of application kernels.

4.1.2 Analytical Modeling

Analytical modeling has long been used in system design, ranging from simple “back of the envelope” computations with a handful of parameters to detailed system models with hundreds of parameters. Analytical models support rapid development, modification, and execution. This makes them particularly beneficial early in the design process to sweep values and identify areas for further research. Also, any one part of the model can be expanded to provide additional detail depending on the area of interest. However, analytical models are often not able to reflect complex interactions, such as congestion in a network, limiting their
accuracy. Although complex models can be developed, the model developer must know that an issue exists and understand the details of how it arises in order to accurately model it.

The remainder of this section presents a simple analytical model implemented in MATLAB and used to evaluate the massively multithreaded architecture. MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation [64]. It is geared toward mathematical computation, allowing the user to model, execute, and analyze problems quickly. To assist in developing algorithms and analyzing data, it provides features such as a high level language with mathematical functions and graphics functions for visualizing data.

An analytical model of system performance is defined in terms of total execution time for an application. It is assumed that each processor in the system is running the same program segment and processing the same workload so that the execution time of the entire program segment is the same as the time that it takes for a single processor to complete its share of the workload.

The performance of an individual processor can be modeled by considering the amount of time it is busy vs. idle. The processor is considered busy when it is executing instructions and idle when there are no operations scheduled that are ready to be executed.

\[ T_{\text{total}} = T_{\text{busy}} + T_{\text{idle}} \]

The busy time can be split into the time spent performing computation and the time spent on overheads such as the creation and destruction of threads and the
handling of parcels.

\[ T_{\text{busy}} = T_{\text{compute}} + T_{\text{overhead}} \]  
\[ T_{\text{overhead}} = T_{\text{thread-mgmt}} + T_{\text{parcel}} \]  

Processor idle time typically occurs when the processor is waiting on long-latency events such as network communication and memory accesses. Multithreading allows instruction execution to be overlapped with network communication and memory access latencies and may also allow the communication and memory latency access latencies of different threads to be overlapped. Therefore, the amount of time that a processor sits idle depends on the ability to overlap these different components of execution time and is a function of both the architecture and the application workload. I represent this as

\[ T_{\text{idle}} = (T_{\text{network}} + T_{\text{memory}})(1 - \eta_{\text{overlap}}) \]  

where, \( \eta_{\text{overlap}} \) is a value between 0 and 1 representing the fraction of the communication and memory access latencies that can be overlapped with other components of execution time. The parameter \( \eta_{\text{overlap}} \) is defined based on characteristics of the application and system architecture.

The remaining components of execution time are a function of both the system level and application level characteristics shown in Table 4.2. Given these parameters I can determine the time spent in each component of execution.

Assume that, when no delays are present, a processor requires an average of one cycle to execute an instruction. Therefore, the computation time for a processor
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>number of processors</td>
</tr>
<tr>
<td>$L_{\text{thread-mgmt}}$</td>
<td>latency involved in managing a thread</td>
</tr>
<tr>
<td>$L_{\text{parcel}}$</td>
<td>latency involved in processing a parcel</td>
</tr>
<tr>
<td>$L_{\text{network}}$</td>
<td>latency involved in traversing the network</td>
</tr>
<tr>
<td>$L_{\text{memory}}$</td>
<td>latency involved in accessing a memory component</td>
</tr>
<tr>
<td>$\tau$</td>
<td>processor clock rate</td>
</tr>
<tr>
<td>$M$</td>
<td>number of threads</td>
</tr>
<tr>
<td>$W$</td>
<td>number of instructions executed for the workload</td>
</tr>
<tr>
<td>$f_{\text{mem}}$</td>
<td>fraction of workload that is memory references</td>
</tr>
<tr>
<td>$p_{\text{remote}}$</td>
<td>probability of a memory reference being remote</td>
</tr>
</tbody>
</table>
is the workload per processor divided by the clock rate.

\[ T_{\text{compute}} = \frac{W/P}{\tau} \quad (4.5) \]

The time spent in thread management is computed as the number of threads per processor times the latency involved in managing a thread. Assuming the threads are uniformly distributed among the processors, this is represented as

\[ T_{\text{thread-mgmt}} = \frac{M}{P} L_{\text{thread-mgmt}} \quad (4.6) \]

Communication time is separated into the processor overhead involved in creating and handling a parcel and the communication costs involved in traversing the network. Communications arise when a remote memory reference is issued.

\[ T_{\text{parcel}} = \frac{W p_{\text{remote}f_{\text{memory}}}}{P} L_{\text{parcel}} \quad (4.7) \]
\[ T_{\text{network}} = \frac{W p_{\text{remote}f_{\text{memory}}}}{P} L_{\text{network}} \quad (4.8) \]

Finally, the time spent accessing memory is computed as

\[ T_{\text{memory}} = \frac{W f_{\text{memory}}}{P} L_{\text{memory}} \quad (4.9) \]

This analytical model is used to efficiently explore performance for a broad range of operating parameters and direct further, more detailed explorations. Results are presented in Section 4.2.1.
4.1.3 Queuing Simulation

A hierarchical queuing model of the system was also developed to extend the results available from the analytical model. The queuing simulation model was developed in Hyperformix Workbench and parameterized to allow extensive sweeps of the design space. Hyperformix Workbench is a hierarchical, transaction oriented, discrete event simulation modeling tool \[45\]. It allows the user to quickly and easily develop, modify, and evaluate a system model at different levels of abstraction. By providing the underlying simulation framework, such as the implementation and management of queues, servers, and event lists, it allows the user to focus on modeling the system at hand. Hierarchical design facilitates the development of a simple system model where individual components of the model can be expanded as the design progresses and additional detail is determined. The graphical user interface with model animation and tracing supports rapid development and debugging. Furthermore, the graphical user interface and animation make the system model useful as a kind of “executable documentation” to coordinate design and ensure that all designers are working from a common understanding of the system. Finally, the automatic collection of statistics and built in analysis tools greatly simplify analysis. Limiting the amount of detail incorporated into the model enables the simulation of massively parallel systems in a reasonable amount of time. However, for very detailed models, custom simulation environments may execute faster.

The primary components of this system model are the lightweight processing chip (LPC), lightweight processor (LWP), memory, and interconnects. Multiple instances of each component can occur in the system model and each component can be defined at varying levels of detail such as a simple delay, a server, a custom
C procedure, or some combination of mechanisms. The remainder of this section describes each of the components of the system.

4.1.3.1 System Level

At the highest level the model consists of one or more LPCs connected by a network. The network is modeled as a simple delay and the number and type of communications is tracked to determine the required bandwidth. Each LPC consists of the following submodels:

- one or more LWPs,
- one or more memories,
- network interface,
- on-chip interconnect.

Communications between LPCs are packaged as parcels and are routed through the network interface. The network interface is divided into input and output server queues with the associated delays representing the overhead involved in handling an incoming or outgoing parcel. The size of the server queues are tracked to determine the buffer sizes needed to handle the traffic on and off a chip.

The LWPs, memory, and network interface are connected by the on-chip interconnect. The on-chip interconnect is modeled as a simple delay and the number and type of transactions are monitored to determine the necessary bandwidth. The LWP and memory components are described in the following sections.
4.1.3.2 Lightweight Processor

Each LWP processor has a frame buffer, instruction cache, and input/output buffers to the on-chip interconnect. The frame buffer consists of a read port and a write port. The ports can be accessed concurrently and each is represented with an associated delay. The instruction cache is also represented as concurrent read and write ports with a delay and an associated miss rate. The input and output buffers for the on-chip interconnect are represented as server queues. Custom C code is associated with the output buffers and is used to determine whether the memory access will be local or remote as well as to generate the address based on the associated parameters. The queue depths are monitored to determine the amount of traffic across the interconnect and the necessary buffer sizes.

The LWP implements processor execution. Each thread of execution is represented as a transaction that flows through the following basic execution cycle:

1. **Thread frame fetch**: A thread frame is fetched (read) from the frame buffer.

2. **Instruction fetch**: An attempt is made to fetch an instruction from the instruction cache. A hit or miss is generated based on the associated parameter. On a hit, an instruction type is generated based on the instruction probabilities determined from the statistical workload (e.g. ALU/CTRL, LOAD, STORE, or SPAWN). Execution then moves on to the next step. On a miss, a memory access is triggered to fetch the instruction and the thread is blocked until the access returns.

3. **Operand fetch**: The registers are read from the current thread’s frame in the frame buffer.

4. **Execute**: The instruction execute phase is represented as a delay parameter.
5. **Memory access**: If the instruction is of type **ALU/CTRL** then there is no functionality associated with this stage. If the instruction is of type **LOAD**, **STORE** or **SPAWN**, then a transaction is split off and sent to memory and the thread becomes blocked. Additional details of the memory functionality are described in the memory section.

6. **Write back**: The write back phase requires a write to the frame cache. This represents the store of a result as well as the update of the thread state.

Each of these is a separate pipeline step so that an instruction from a different thread can be at each stage of the pipeline concurrently.

The blocking of threads and management of thread frames is controlled from the LWP as well. When a memory operation occurs, a transaction is split. The new transaction moves out to the memory system to execute the load or store. The original transaction then moves on to a thread control node to determine whether the thread blocks and, if so, if it needs to be evicted to memory. The transaction may move on to block “in the frame buffer” or may be evicted to block “in memory” allowing a new frame to be brought in. When a memory operation completes, it releases the associated blocked transaction. Additional details are found in the frame management section.

4.1.3.3 Memory

Each LPC has an associated memory component. At the most general level, a memory component can be modeled as an average memory access latency. At a more detailed level, each memory component is defined further to include:

- one or more eDRAMs
• one or more DRAMs

• memory interconnect

The on-chip memory consists of embedded DRAM (eDRAM). The LWPs, eDRAMs, and DRAMs are treated as uniform pools of resources so that any LWP can access any eDRAM via the on-chip interconnect and any eDRAM can access any DRAM associated with that chip via the memory interconnect. Remote memories associated with another LPC must be accessed by sending a parcel via the network.

The eDRAMs and DRAMs are represented as delays that depend on the request type. For the eDRAMs all requests of 256 bytes take 11 cycles fixed and those of 32 bytes take 4 cycles fixed. This comes from an assumption that 32 bytes are delivered in 4 cycles from a random access and then the rest of the line is delivered each cycle for the next 7 cycles to get the 256 contiguous bytes. The 4 cycle requests are for instructions and acknowledgements with the rest being 11 cycles. Only one memory access is handled at a time and there is a queue associated with each eDRAM. The DRAM is modeled the same way using delays of 50 cycles for 32 bytes and $50 + 2 \times 7 = 64$ cycles for 256 bytes. The eDRAM has a maximum bandwidth of approximately $256/11 = 23.27$ bytes/cycle. The DRAM has a maximum bandwidth of approximately $256/64 = 4$ bytes/cycle. The memory queue depths and request types are monitored to determine the ability of the memories to handle the memory traffic. The memory interconnect is represented as a single cycle delay and the traffic is monitored to determine the bandwidth requirements.
4.1.3.4 Thread Frame Management

Thread frames may be located on the active list of ready threads, in an LWP’s frame buffer, or on the blocked list represented by the frame block node. Two additional submodels are specified to enable thread frame management: the active list and the frame block node. Although these are not physical components of the architecture, separate submodel components are used to represent them.

Frame management is handled primarily through the active list submodel. An allocate node represents the active frame list in memory with a resource node used to keep track of the number of available frame slots on the LPC via tokens (resource tokens = frame buffer size × number of LWPs). If there is a token available, that frame is chosen for execution and gets sent to an LWP with an available frame slot in its frame buffer. When a frame is replaced on the active frame list (i.e. it is no longer in a frame buffer), it releases the token that it held. Frames are initially randomly assigned to the LWPs and memory accesses are random across the on-chip and off-chip memories. The active list is assumed to be uniformly distributed throughout memory and serves all the LWPs.

The frame block node holds those frames (transactions) that are blocked in memory. When a frame blocks, it releases the token that it held so that a new frame from the active list can be fetched into the LWP’s frame buffer. When the memory operation on which the frame is blocked is completed, it “wakes up” the frame using a data resource that the two transactions share. On wake-up, a frame blocked in memory is placed at the end of the active list.
4.1.3.5 Workload Generation

The queuing simulation model is used to address various design issues including the details of the processor microarchitecture, ratio of processing to memory, thread management issues, bandwidth issues, and parcel issues. Key architectural parameters include memory access latencies (both on-chip and off-chip), thread management overheads, network bandwidth and latency, memory system configuration, and number of processing nodes. The probability of each instruction type, and number of threads in the system are the key application parameters explored.

Once the key parameter distributions have been defined, they become model parameters and are used to drive the statistical simulation. Because statistically identical threads are used, the instruction trace is generated dynamically during execution. The simulation begins with some number of threads available in the system. At each cycle a thread is chosen for execution with the instruction cache miss rate used to determine whether the instruction fetch is a hit or miss and the associated latency. Next, the instruction type is generated from one of the following categories based on the instruction distributions: ALU/CTRL, LOAD, STORE, SPAWN, and TERMINATE. If the instruction is a memory operation, the memory address distribution is used to determine if the reference is on-chip or off-chip. The frequency of the TERMINATE instruction determines thread length while the SPAWN frequency determines whether the average number of threads in the system is increasing, steady, or decreasing.

4.2 Evaluation of Blocking vs. Non-blocking Memory References

Should a thread block when it issues a memory request, or should it be allowed to issue additional instructions, including memory references, blocking only if the
result is not available when needed? This decision affects not only the availability of threads to issue instructions but also the amount of memory traffic generated by swapping frames between memory and the frame buffer. This question is explored in the context of a particular application, the simple yet highly parallel problem of calculating the sum of two vectors of size $n$ as presented in [17]. A multithreaded implementation of this problem distributes the work by spawning $m$ independent threads to add different regions of the arrays and Figure 4.1 shows the assembly code for the loop within each thread. This core loop consists of six instructions, half of which require a memory access and define the instruction probabilities for the model parameters.

```
loop:
    load  a, A[i]
    load  b, B[i]
    add   c, a, b
    store C[i], c
    incr  i
    branch i<n, loop
```

Figure 4.1. Assembly Code for Vector Sum [17, page 2]

Figure 4.2 illustrates the timing of the code fragment for a single thread for the cases of both blocking and non-blocking loads. The latency of a memory access is arbitrarily shown as 5 cycles. The first column shows the timing for the case of a blocking load. Here, the next instruction in the sequence cannot issue until the
memory operation completes. The second column illustrates the time for a non-blocking load. In this case, instructions continue to issue past the load until an instruction is reached that requires the result of a load, at which point the thread blocks. Ideally, this allows the latency of the two loads to be entirely overlapped without increasing the latency.

<table>
<thead>
<tr>
<th>Blocking Load</th>
<th>Non-blocking Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>(block before issuing next instruction)</td>
<td>(block before using result)</td>
</tr>
<tr>
<td>load a, A[i]</td>
<td>load a, A[i]</td>
</tr>
<tr>
<td>stall</td>
<td>load b, B[i]</td>
</tr>
<tr>
<td>stall</td>
<td>stall</td>
</tr>
<tr>
<td>stall</td>
<td>stall</td>
</tr>
<tr>
<td>stall</td>
<td>stall</td>
</tr>
<tr>
<td>load b, B[i]</td>
<td>add c, a, b</td>
</tr>
<tr>
<td>stall</td>
<td>store C[i], c</td>
</tr>
<tr>
<td>stall</td>
<td>incr i</td>
</tr>
<tr>
<td>stall</td>
<td>branch i&lt;n, loop</td>
</tr>
</tbody>
</table>

Figure 4.2. Blocking versus non-blocking loads [17, page 3]

When considering a single thread, non-blocking loads have only half the stall cycles of blocking loads. This fact is less important in a multithreaded machine, however, since other threads can execute during the delay, effectively hiding the latency. An important question therefore is how many threads are needed to keep
the processor busy while the loads are outstanding in either case. In general, the critical number of threads required to keep the processor busy can be represented as:

\[ M_{\text{crit}} = \frac{L}{b} \] (4.10)

where, \( M_{\text{crit}} \) is the critical number of threads needed to hide latency, \( L \) is the average latency of a memory reference, and \( b \) is the number of independent instructions between a load and the use of its result.

Equation 4.10 implies that one could reduce the number of threads required to keep a single multithreaded processor busy by optimizing the program, using techniques such as loop unrolling, to increase the number of instructions between a load and the use of its result. However, is there any advantage to restructuring the code versus creating additional threads? The answer to this question depends on many factors, including the cost of multithreading and the cost of supporting multiple outstanding loads per thread, as well as other complex interactions that occur. Two sets of experiments are used to address this question. First, a simple analytical model is used to gain insight into the key trends, then a queuing simulation model is used to provide a more detailed analysis.

The remainder of this section explores the question of whether a thread should block when issuing a memory reference or whether the system should be configured to support non-blocking memory references which allow the thread to continue execution after issuing a memory reference, blocking only if the result of the memory reference is not available when needed. Figure 4.3 shows the high level architectural model used for this set of experiments. It focuses on system wide effects, representing each node as a single processor and memory with the memory modeled as an average memory access latency. A simple analytical model derived
from [17] provides insight into the potential performance provided by non-blocking memory references. Then a queuing simulation model gives a more detailed analysis. Results show that non-blocking memory references generally improve parallel speedup by reducing the number of threads needed to keep the processor busy. However, in cases where the memory system is unable to handle the memory traffic generated, allowing multiple outstanding memory references per thread or providing additional threads are unable to improve performance. Memory latency and bandwidth are the dominant parameters affecting parallel speedup.

![Diagram of Architectural Model for Evaluation of Blocking vs. Non-blocking Memory References](image)

**Figure 4.3.** Architectural Model for Evaluation of Blocking vs. Non-blocking Memory References

4.2.1 Analytical Modeling Results

Using the analytical model described, a set of parameter sweeps are done to explore the relationship between performance and the number of processors, number of threads, thread overhead, communication and memory latencies, and blocking behavior for ranges of interest. The analytic model is able to efficiently
evaluate large systems with long latencies. Therefore, many of the values chosen for this model are larger than those used for the queuing-based model, discussed later in this chapter. Table 4.3 summarizes the parameters and their default values for the analytic model. The architectural parameters are chosen based on discussions with Cascade system architects to represent areas of interest. The application parameters are estimated based on the vector sum application used to drive the analysis. For each element of the vector, the main loop of the assembly code contains six instructions: three ALU/CTRL operations, two LOADs, and one STORE. Therefore the workload consists of six instructions per element and half of those are memory references. Each of the $P$ processors is assumed to have an associated memory and the data elements are uniformly distributed throughout the memories so that the likelihood of an access being on any given memory is $1/P$.

An additional parameter is added to the model to explore the effects of allowing multiple outstanding memory loads. The parameter $b$ corresponds to the number of outstanding memory loads a thread is allowed to issue before it must block. When $b = 1$ a thread is allowed to issue only one memory reference and then must block. Two loads are allowed when $b = 2$ and so on.

Two assumptions alter the computation of the idle time. First, I assume that the network communication and memory access latencies of threads on the same processor can be overlapped so that the total latency is equivalent to that of a single thread. Therefore the total network communication and memory access time is divided by the number of threads per processor. Second, in Equation 4.10, $M_{\text{crit}}$ is defined as the number of threads required to hide the latency of a memory access. Therefore $\eta_{\text{overlap}}$, the degree to which processor execution can be
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P )</td>
<td>number of processors</td>
<td>( 2^{10} )</td>
</tr>
<tr>
<td>( L_{\text{thread-mgmt}} )</td>
<td>latency involved in managing a thread</td>
<td>2</td>
</tr>
<tr>
<td>( L_{\text{parcel}} )</td>
<td>latency involved in processing a parcel</td>
<td>1</td>
</tr>
<tr>
<td>( L_{\text{network}} )</td>
<td>latency involved in traversing the network</td>
<td>256</td>
</tr>
<tr>
<td>( L_{\text{memory}} )</td>
<td>latency involved in accessing a memory component</td>
<td>4</td>
</tr>
<tr>
<td>( \tau )</td>
<td>processor clock rate</td>
<td>1 GHz</td>
</tr>
<tr>
<td>( b )</td>
<td>operations issued between a load and use</td>
<td>1</td>
</tr>
<tr>
<td>( M )</td>
<td>number of threads</td>
<td>( 2^{16} )</td>
</tr>
<tr>
<td>( N )</td>
<td>number of elements per vector</td>
<td>( 2^{30} )</td>
</tr>
<tr>
<td>( W )</td>
<td>number of instructions executed for the workload</td>
<td>( 6N )</td>
</tr>
<tr>
<td>( f_{\text{mem}} )</td>
<td>fraction of workload that is memory references</td>
<td>0.5</td>
</tr>
<tr>
<td>( p_{\text{remote}} )</td>
<td>probability of a memory reference being remote</td>
<td>( (P - 1)/P )</td>
</tr>
</tbody>
</table>
overlapped with communication and memory access latencies, is equivalent to the ratio of the number of threads on a given processor to $M_{\text{crit}}$. Combining these two sets of assumptions yields the following equation for $T_{\text{idle}}$:

$$T_{\text{idle}} = \frac{T_{\text{network}} + T_{\text{mem}}}{M/P} (1 - \frac{M/P}{M_{\text{crit}}}). \quad (4.11)$$

Figure 4.4 shows how the total execution time varies with the network communication latency for $b = 1$ and $b = 2$ with all other parameters held constant. For blocking loads ($b = 1$) the maximum latency that can be hidden is $M/P = 64$ cycles. As long as the latency is less than that, there is no idle time and the execution time is independent of latency. However, when the latency exceeds 64 cycles, there are no longer enough threads to keep the processor fully utilized and the execution time increases linearly with the network latency. With two outstanding loads per thread ($b = 2$), the system is able to hide a latency that is twice as long before the performance becomes limited by communication. In general, the “knee” of the curve will move to the right as the number of outstanding loads is increased, as predicted by Equation 4.10.

Next I investigate the effect on execution time of varying the network communication latency and the number of threads. Figure 4.5 shows the results. As the number of threads increases, higher network latencies can be masked and the execution time decreases. In order to fully mask the latency, $2^{18}$ threads are needed. Increasing the number of threads up to that point, results in decreased execution time. Additional threads beyond that point, however, have no impact on performance.

Figure 4.6 shows how the parallel speedup varies with the number of processors and network communication latency. For a given latency there are two distinct
Figure 4.4. Total execution time versus network communication latency for blocking vs. non-blocking memory accesses in analytical model
Figure 4.5. Total execution time versus network communication latency for varied number of threads
regions on the speedup curve: a compute-limited region where the processors are fully utilized and a communication-limited region where the processors are starved. The transition between the two regions occurs when the number of threads per processor drops below the critical number needed to fully mask the network communication latency. As long as the processors are kept busy, latency has no effect on speedup because it can be fully masked by the computation. As the network latency increases, more threads are needed per processor to mask it and, for a fixed number of threads, performance becomes communication-limited with fewer processors. Adding additional processors in the communication-limited region has little impact on performance.

Figure 4.6. Speedup versus number of processors for varied network communication latency
As stated in Equation 4.10, the number of threads per processor required to hide the average latency of a memory reference is inversely proportional to the number of instructions, $b$, that can be issued between a load and the use of its value. In the vector addition example, $b$ can be increased using loop unrolling. In order to take advantage of this, however, both the processor and the memory system must be able to support $b$ outstanding loads per thread concurrently. Figure 4.7 shows the effect on parallel speedup of non-blocking loads. As the number of outstanding loads increases, the transition from the compute-limited to communication-limited region occurs at a larger number of processors. However, after the transition, performance converges to the same level independently of $b$.

Figure 4.7. Effects of number of outstanding memory references on speedup
This analytical model provides valuable insights into the system performance and the effects of allowing non-blocking memory accesses. Two distinct operating regions are identified with each exhibiting different performance characteristics. In the communication-limited region there are not enough threads available to mask the communication latency and the processor spends time idle waiting for the results of memory accesses. In the computation-limited region, there are enough threads available to issue an instruction in every cycle, keeping the processor fully saturated, and communication latency has no effect on performance. In general, the equation $M_{\text{crit}} = L/b$ predicts the transition between these two regions as the critical number of threads needed to saturate the processor. Supporting non-blocking memory references allows the system to mask larger latencies with the same number of threads. This reduces the number of threads needed to saturate the processor. For a constant number of threads, this causes the transition between the compute-limited and the communication-limited region to occur at a larger number of processors. Within the compute-limited region, however, supporting non-blocking memory references or increasing the number of threads has no effect.

Although only the network communication latency was shown, similar effects are seen for the memory access latency with the current model. Parallel speedup depends heavily on the ability to mask the memory reference latencies and keep the processor busy. As the communication latency increases, this increases the number of threads per processor needed to mask the latency and increases the overall execution time in the communication-limited regions. Supporting multiple outstanding memory references reduces the number of threads needed to mask a given latency. One of the limitations of this model is that it assumes a uniform network latency and memory access latency and cannot represent network con-
tention or queueing at the memory that may occur with increased memory traffic. This is especially important in exploring the effects of multiple outstanding memory references because this increases the number of references allowed per thread and thus the total memory traffic. The next section describes a queuing simulation model derived from the analytical model. This model is used to provide more detailed simulations to verify and expand on the analytical modeling results.

4.2.2 Queuing Simulation Results

The queuing simulation model is derived from the analytical model. In order to reduce simulation size and execution time, however, the queuing simulation uses smaller values for some of the default parameters, such as the number of processors and number of threads, and each processor’s memory is modeled as a single server with an average access latency. Table 4.4 lists the key simulation parameters and their default values. As was done for the analytical model, the architectural parameters were chosen through discussion with various Cascade system architects and the application characteristics are based on the vector sum application. In order to facilitate direct comparison between the analytical modeling and the queuing-based simulation, some of the analytical analysis was repeated using these parameters.

The simulation model provides significantly more memory system detail than that provided by the analytical model. The simulation models not only the latency involved in communication and memory accesses but also the queueing of requests at memory and the bandwidth required by the communication network. Experiments are used to explore the effectiveness of non-blocking memory accesses in hiding both the memory access latency and the network communication latency.
### Table 4.4

**Default Queuing Simulation Parameter Values for Evaluation of Non-Blocking Memory References**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>number of processors (1 per LPC)</td>
<td>varies</td>
</tr>
<tr>
<td>$L_{\text{thread-mgmt}}$</td>
<td>latency involved in managing a thread</td>
<td>2</td>
</tr>
<tr>
<td>$L_{\text{parcel}}$</td>
<td>latency involved in processing a parcel</td>
<td>1</td>
</tr>
<tr>
<td>$L_{\text{network}}$</td>
<td>latency involved in traversing the network</td>
<td>2</td>
</tr>
<tr>
<td>$L_{\text{memory}}$</td>
<td>latency involved in accessing a memory component</td>
<td>4</td>
</tr>
<tr>
<td>$\tau$</td>
<td>processor clock rate</td>
<td>1 GHz</td>
</tr>
<tr>
<td>$b$</td>
<td>operations issued between a load and use</td>
<td>2</td>
</tr>
<tr>
<td>$M$</td>
<td>number of threads</td>
<td>varies</td>
</tr>
<tr>
<td>$N$</td>
<td>number of elements per vector</td>
<td>$2^{14}$</td>
</tr>
<tr>
<td>$W$</td>
<td>number of instructions executed for the workload</td>
<td>$6N$</td>
</tr>
<tr>
<td>$f_{\text{mem}}$</td>
<td>fraction of workload that is memory references</td>
<td>0.5</td>
</tr>
<tr>
<td>$p_{\text{remote}}$</td>
<td>probability of a memory reference being remote</td>
<td>$(P - 1)/P$</td>
</tr>
</tbody>
</table>
The first component of the memory reference latency to be explored using simulation is the memory access latency. Figure 4.8 shows the variation in execution time as the memory access latency and number of outstanding memory references are increased while both the number of processors (P=16) and number of threads (M=64) are held constant. Unlike the results shown by the analytical model, in the queuing simulation there is no compute limited region because the memory almost immediately becomes a bottleneck. This arises from the collision of multiple memory references to the same memory, which is not accounted for in the analytical model. The analytical model assumes that the communication and memory access latencies of all threads on a processor can be overlapped. However, this is only possible if the memory system is able to support one memory reference per cycle per processor.

How does this bottleneck occur? Consider the assembly code for the vector sum program; half of the instructions are memory references. If an ALU operation takes one cycle, there is a memory reference issued from each thread, on average, every 2 cycles. However, the minimum memory access time is 4 cycles. If two successive accesses are to the same memory bank, the second one must wait 2 cycles before it can even begin. Therefore, the memory requests begin to queue up at the memories resulting in increasing response times for memory access. Increasing the number of threads further increases the number of queued memory accesses compounding the problem (up to the point where the processor is fully utilized).

Table 4.5 shows the processor and memory performance characteristics for a simulation with two non-blocking loads, P = 16, and M = 64 or 256. For a latency of 4 cycles and 64 threads, there are on average 5 memory references queued
Figure 4.8. Total execution time versus memory access latency for blocking vs. non-blocking memory accesses
at a given memory and memory utilization is 90%. As the latency increases, the number of requests queued up at memory increases and more threads are blocked, making fewer threads available for execution. This is reflected by increased utilization of the memory and decreased processor utilization. These performance characteristics demonstrate that the memory becomes a bottleneck even for small latencies and worsens as the memory latency increases.

### TABLE 4.5

**PROCESSOR AND MEMORY PERFORMANCE CHARACTERISTICS FOR VARIED MEMORY ACCESS LATENCY**

\( (P = 16, b = 2) \)

<table>
<thead>
<tr>
<th>Performance Characteristic</th>
<th>Threads</th>
<th>Memory Access Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Memory Queue</td>
<td>64</td>
<td>5.490</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>21.38</td>
</tr>
<tr>
<td>Active List Queue</td>
<td>64</td>
<td>0.1473</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>1.231</td>
</tr>
<tr>
<td>Memory Utilization</td>
<td>64</td>
<td>0.9035</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>0.9461</td>
</tr>
<tr>
<td>Processor Utilization</td>
<td>64</td>
<td>0.2713</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>0.2837</td>
</tr>
</tbody>
</table>
In a multithreaded system, increasing the number of threads typically improves performance by providing more threads to mask latencies. Table 4.5 demonstrates that increasing the number of threads does make more threads available for execution on the active list and provides a slight increase in processor utilization. It also increases the congestion in memory, however, which results in longer memory access latencies.

Figure 4.9 demonstrates the effect on execution time of varying memory latency and the number of threads for a constant number of processors \((P = 256)\). For a system with unlimited bandwidth, the execution time continues to decrease as the number of threads increases due to a larger pool of instructions from which to issue. When bandwidth issues come into play, however, this improvement is tempered by an increase in the number of outstanding memory references and thus the memory response time. This results in slightly greater execution times as the number of threads exceeds 1024. In general, it is important to have enough threads to keep all the processors busy but too many threads issuing memory references adds to the bottleneck at memory and results in a decrease in performance.

Figure 4.10 examines the effects on parallel speedup of varying the number of processors and the memory access latency while the number of threads are held constant at \(M = 1024\). In general, as either the latency or the number of processors increases, the number of threads needed to saturate the system increases as well. This is reflected in the analytical modeling results. For the simulation, however, a number of additional factors come into play. Increasing the number of processors improves performance by providing more concurrent execution and, because I assume that each processor has an associated memory, increasing the total memory available in the system thereby reducing contention. However, more
Figure 4.9. Effects on execution time of varying the memory access latency and number of threads.
processors also results in fewer threads available at each processor to mask latencies and allows more memory references to be issued concurrently. Overall, these factors balance each other, and parallel speedup continues to improve as the number of processors is increased.

Many of the differences between the expected performance indicated by the analytical model and the simulation performance are due to the large fraction of memory references in the vector sum application and the resulting memory congestion. An instruction mix with a higher ratio of alu operations to memory operations reduces the memory contention effects and should yield results that more closely resemble those of the analytical model. It is possible to approximate the instruction mix necessary to mask a given memory latency on a set number of processing nodes. Assume that the latency of each ALU operation, \( A \), is 1 cycle. Also assume that each local memory access is \( L_{\text{memory}} \) cycles and each remote memory access is \( 2(L_{\text{memory}} + L_{\text{network}}) \) cycles (i.e. one memory access and one network communication latency for the request and one each for the response). Therefore, setting the computation time equal to the memory reference latencies yields the following equations:

\[
\begin{align*}
    f_{\text{alu}} \times A &= f_{\text{mem}}(p_{\text{local}}L_{\text{memory}} + p_{\text{remote}}2(L_{\text{memory}} + L_{\text{network}})) \quad (4.12) \\
    f_{\text{alu}} &= (1 - f_{\text{alu}})(\frac{1}{P}L_{\text{memory}} + \frac{P-1}{P}2(L_{\text{memory}} + L_{\text{network}})) \quad (4.13) \\
    f_{\text{alu}} &= \frac{L_{\text{memory}} + 2(P-1)(L_{\text{memory}} + L_{\text{network}})}{L_{\text{memory}} + 2(P-1)(L_{\text{memory}} + L_{\text{network}}) + P} \quad (4.14)
\end{align*}
\]

Assume that I wish to mask a memory access latency of 16 cycles on 16 processors. Using \( L_{\text{memory}} = 16 \), \( P = 16 \), and the default value of \( L_{\text{network}} = 2 \) yields \( f_{\text{alu}} = 0.97 \) and \( f_{\text{mem}} = 0.03 \). Assuming two-thirds of the memory accesses are
Figure 4.10. Speedup versus number of processors for varied memory access latency.
LOAD instructions and one-third STORE instructions, this results in a computation intensive instruction mix of 97% ALU, 2% LOAD and 1% STORE.

Figure 4.11 shows the execution time for a simulation with the computation intensive instruction mix defined above, increasing memory access latency, and the number of processors and threads held constant at P=16, M=64. These results more closely resemble those of an unlimited bandwidth system, with a “knee” in the curve at a memory access latency of 16. Increasing the number of outstanding memory references allowed before a thread blocks decreases the total execution time. The most significant reduction occurs between $b = 1$ and $b = 2$ with little change occurring for values of $b$ greater than 2.

![Figure 4.11. Execution time vs. memory access latency for computation intensive instruction mix](image)
Next I explore the network communication latency. I assume the network can handle one parcel per processor per cycle and network congestion is not modeled by the simulation. This yields results that more closely mimic those of the analytical model. Monitoring the number of transactions across the network identifies the bandwidth required to support this assumption.

Figure 4.12 shows the effects on execution time of increasing network latency and allowing multiple outstanding memory references ($P = 16, M = 1024$, vector sum instruction mix). Allowing multiple outstanding memory references increases the number of instructions that a thread can issue before it blocks and thus the amount of latency it can hide. This results in a corresponding reduction in the execution time for increasing values of $b$. The execution time stays fairly constant for network latencies up to 16 when $b = 1$. This agrees with the analytical model’s statement that the maximum latency that can be hidden is approximately $M/P$ cycles. As the number of outstanding memory references is increased, the network latency that can be masked increases as well, moving the “knee” of the curve to a network latency of 64 for $b = 2$ and $b = 4$, and to a network latency of 256 for $b = 8$. Larger network latency results in a corresponding increase in the execution time. However it does not cause contention at the memories as seen with increased memory access latency.

Additional insight into these effects can be seen by examining the queue depths for the memory and active list. Table 4.6 presents the mean queue depths and execution time for the same simulation with $b = 2$. As the network latency increases, more threads are blocked waiting for the result of a memory access thereby reducing the number of threads available on the active list and reducing memory contention.
Figure 4.12. Execution time for increasing network communication latency and number of outstanding memory references

TABLE 4.6

MEAN QUEUE POPULATION FOR VARIED NETWORK DELAY

\( (P = 16, M = 1024, b = 2) \)

<table>
<thead>
<tr>
<th>Network Delay</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Queue</td>
<td>58.79</td>
<td>28.93</td>
<td>25.02</td>
<td>21.72</td>
<td>0.6665</td>
</tr>
<tr>
<td>Active List Queue</td>
<td>3.801</td>
<td>3.387</td>
<td>3.29</td>
<td>2.8</td>
<td>0.8097</td>
</tr>
</tbody>
</table>
Next I investigate the effect on execution time of varying the number of threads for a range of network latencies ($P = 256, b = 2$). The number of threads was varied from 256 to 16384 and the results are shown in Figure 4.13. Increasing the number of threads improves performance up to the point where the system begins to become compute-limited. Increasing the number of threads past this point adds to congestion. For small network latencies, large numbers of threads increase the number of memory references to the point where the memory becomes a bottleneck and performance suffers. More threads are needed to mask larger network latencies so increasing the number of threads provides continued improvement in performance.

![Figure 4.13. Effects on execution time of varying the network latency and number of threads.](image-url)
Figure 4.14 shows the parallel speedup with increasing numbers of processors for network latencies from 1 to 1024 ($M = 1024, b = 2$). These results are very similar to those of the analytical model. For a given network latency the number of threads available to mask the latency decreases as the number of processors increases. For small network latencies the number of threads available is sufficient to mask most of the latency up to 256 processors whereas for larger latencies there is a transition point at which the number of threads available is not enough to effectively mask the latency and the speedup begins to plateau. This transition point moves to smaller numbers of processors as the latency increases. Overall, the speedup decreases as the network latency increases because more threads are needed to mask the increased latency.

Allowing multiple outstanding memory references can reduce the number of threads needed to mask latency. However, performance is very closely tied to the latency and bandwidth of the communication and memory system. Ideally, the system should be able to handle one memory operation per processor per cycle for memory intensive applications. When this is not the case, the application’s instruction mix affects performance. In general, adding more threads enables the system to mask greater latencies. However if the threads are memory intensive, additional threads increase the network bandwidth requirements and contention at memory and can increase the overall memory reference latency. Adding additional processors increases the number of memory requests that can be issued concurrently, but also distributes those accesses over more memories. Parallel speedup is greatly dependent on the network communication delay and the number of threads available to mask it. The number of threads available for execution may be increased by increasing the number of threads in the system or by in-
Figure 4.14. Speedup versus number of processors for varied network latency.
creasing the number of outstanding memory references allowed before a thread blocks.

4.3 Evaluation of Thread Frame Eviction Policy

When a thread blocks, it can no longer be used to issue instructions and keep the processor busy. An eviction policy determines if and when a blocked thread frame should be swapped from the frame buffer to memory to allow a new thread frame to be loaded. Eviction policies may range from immediately evicting a blocked thread to evicting a blocked thread when there are no longer enough active threads available in the frame buffer to keep the processor fully utilized. Immediate eviction ensures that the buffer is full of active threads (when such threads are available), making it more likely the processor will remain fully utilized and increasing the number of threads executing “concurrently” in the system. However, it is also likely to result in more evictions and thus more memory traffic, possibly increasing the overall latency of a swap. Waiting until a thread must be evicted (i.e. there are not enough active threads in the frame buffer to keep the processor busy) may reduce the total number of evictions and thus the memory traffic and other overheads associated with the transfer. However, it may also cause the processor to sit idle waiting for a new thread to be swapped in from memory rather than allowing processor execution to overlap with the latency of evicting a blocked thread and bringing in a new thread.

The efficiency of a given eviction policy depends on a number of factors including the cost of evicting a thread, the number of threads available, and the frequency and latency of blocking. There may also be secondary effects on performance with frequent evictions such as increased instruction cache miss rates or
extended memory latencies because of the additional memory traffic. In order to explore these issues, a simple eviction policy is defined that specifies a limit on the fraction of the threads in a frame buffer that can be blocked. Any blocked thread that causes the limit to be exceeded is evicted. The value of this limit can easily be swept from 0, corresponding to the case where blocked threads are automatically evicted to memory, to 1, where all threads in the buffer must be blocked before an eviction occurs. In addition to the blocking limit, a number of other configuration parameters are explored including the number of processors, number of threads, number of on-chip memory banks, instruction distribution, and instruction cache miss rate.

This approach chooses to evict the most recently blocked thread for two reasons. First, it simplistically assumes that if all threads block for approximately the same amount of time, then evicting a thread immediately means that it will be blocked longer than a thread that has already been waiting. Second, evicting the current thread reduces the overhead involved in choosing a thread to evict. More complicated, and likely better performing, heuristics could be used for choosing when to evict a thread and which thread to evict. However, this simple eviction policy provides a reasonable design point from which to investigate the system-wide effects of the frequency of eviction and determine the configuration needed to support a given level of evictions.

This section explores the thread frame eviction policy which determines when to evict a blocked thread’s frame from the frame buffer to memory and allow an available thread to be swapped in. The eviction policy monitors the number of blocked threads in the frame buffer, evicting any thread that would increase the number of blocked threads beyond a predefined limit. A queuing simulation
model evaluates the system bandwidth needs under various configurations. In particular, the effect of the eviction policy on processor utilization as well as the bandwidth requirements of the interconnects, on-chip memory, and off-chip memory are assessed. The analytical model is not used to evaluate the thread eviction policy because it does not adequately capture the complex interactions within the memory system that may significantly affect performance.

In contrast to the previous set of experiments which focused on multiple nodes modeled as a single processor and memory per node, the model used here focuses on a single node and its configuration. Each node consists of an LPC and its associated off-chip DRAM as described previously in Figure 3.2. Varying numbers of LWPs and eDRAM banks per LPC are explored. The model assumes that the on-chip embedded DRAM (eDRAM) is used as a cache and that there are four DRAM chips per LPC. Memory access times for both eDRAM and DRAM are computed based on the values presented previously. A thread is assumed to block when it issues a memory reference. Table 4.7 summarizes the key parameters and their associated range of values. The workload is composed of statistically identical threads of infinite length. The number of threads in the workload is varied to represent four different cases:

- 8 threads: low number of threads where all active threads fit into the frame buffer
- 128 threads: number of threads equal to the size of the frame buffer
- 256 threads: moderate number of threads that slightly exceeds the size of the frame buffer
- 1024 threads: large number of threads so that there are always threads
available on the active list

Two different instruction mixes are used to represent different classes of applications of interest in Cascade: a memory intensive mix (MI) of 40% ALU, 40% LOAD, 20% STORE and a compute intensive (CI) mix of 65% ALU, 26% LOAD, 9% STORE.

**TABLE 4.7**

THREAD FRAME EVICTION POLICY PARAMETERS AND ASSOCIATED SWEEP RANGES

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Sweep Range</th>
<th>MI</th>
<th>CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>number of processors per LPC</td>
<td>1 – 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$eDRAM$</td>
<td>number of embedded DRAM banks per LPC</td>
<td>4 – 256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$DRAM$</td>
<td>number of DRAM chips per LPC</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$M$</td>
<td>number of threads</td>
<td>8 – 1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_{alu}$</td>
<td>fraction of workload that is ALU ops</td>
<td>0.4 0.65</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_{load}$</td>
<td>fraction of workload that is LOAD ops</td>
<td>0.4 0.26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f_{store}$</td>
<td>fraction of workload that is STORE ops</td>
<td>0.2 0.09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_{miss}$</td>
<td>probability of an instruction cache miss</td>
<td>0.05, 0.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F_{limit}$</td>
<td>number of blocked frames allowed in the frame buffer</td>
<td>0, 32, 64, 96</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first set of experiments investigates execution under a eviction limit of
0. This causes all frames to be immediately evicted to memory when blocked, allowing a new frame to be swapped into the frame buffer from the active list. A thread is allowed to block in the frame only if there are no threads available on the active list to replace it. The instruction miss rate is set to 5\% and a memory intensive (MI) instruction mix is used. The main performance measures include the issue rate, defined as the fraction of cycles that a processor is able to issue an instruction, the mean queue depth at an eDRAM or DRAM memory, and the mean bandwidth at an eDRAM or DRAM memory.

Figure 4.15 shows both the issue rate and the mean eDRAM queue length for a system with 4 eDRAMs per LPC as the number of threads per processor and number of processors is varied. The eDRAMs quickly become a bottleneck with eDRAM queue depths reaching almost one thousand for 16 processors. As the number of processors increases, the congestion at memory increases causing more threads to block and a corresponding reduction in the issue rate. Therefore, adding additional processors does not improve performance if the memory system cannot handle the additional traffic and it is important to have protocols in place to prevent large queues. Adding more threads per processor effects the system in two ways: by increasing the number of threads that are available to mask the memory latency as well as by increasing the number of memory references. The issue rate improves when increasing from 8 to 128 threads per processor because the effect of having more threads to mask memory latency outweighs the additional memory latency. However, when the number of threads exceeds the size of the frame buffer (128) then additional memory traffic is generated by the swapping of frames between the frame buffer and memory and the issue rate decreases. Therefore, for 4 eDRAMs per LPC, the latency hiding effects found by increasing
the number of threads per processor is effective at hiding the additional memory traffic from memory instructions but not sufficient to mask the additional frame traffic when the frame buffer size is exceeded.

Increasing the number of eDRAMs to 256 reduces the congestion at the eDRAMs and greatly improves the issue rate as shown in Figure 4.16. The mean queue depths are reduced to less than 1 and the issue rate reaches 100% for up to two processors and 1024 threads. Although the congestion at the eDRAMs has been reduced, increasing the number of eDRAMs to 256 increases the pressure on the DRAMs causing falling issue rates for greater than two processors due to congestion at the DRAMs. When there is no congestion in the memory (eDRAM or DRAM), issue rates of 100% can be reached with no adverse effects from increasing the number of processors or increasing the number of threads per processor.

Given the importance of the memory system in performance, I wish to explore the number of eDRAMs and bandwidth per eDRAM needed to support a processor. Figure 4.17 shows the issue rate and mean eDRAM bandwidth for 1 processor with increasing numbers of threads and eDRAMs. Sixty-four eDRAM are needed to support up to 1024 threads per processor while 16 eDRAM are sufficient to support up to 128 threads per processor. These results hold for up to four processors per LPC; after that point the DRAM begins to become a bottleneck, adversely affecting performance. For 64 eDRAMs per LPC, this requires a mean bandwidth of 10 bytes/cycle per eDRAM, approximately half the peak bandwidth of 23 bytes per cycle, and 1.2 bytes/cycle per DRAM, approximately one third of the peak bandwidth of 4 bytes/cycle. These bandwidth requirements increase as the number of processors and/or number of threads per processor increases.
Figure 4.15. Issue Rate and eDRAM Queue Depth for 4 eDRAMs per LPC (Eviction Limit = 0, Memory Intensive Instruction Mix, Instruction Miss Rate = 5%)
Figure 4.16. Issue Rate and eDRAM Queue Depth for 256 eDRAMS per LPC (Eviction Limit = 0, Memory Intensive Instruction Mix, Instruction Miss Rate = 5%)
Figure 4.17. Issue Rate and Mean eDRAM Bandwidth for increasing number of eDRAMS per LPC (Eviction Limit = 0, Memory Intensive Instruction Mix, Instruction Miss Rate = 5%)
Although this is not directly modeled by the simulation, one possible side effect of frequently swapping threads could be an increase in the instruction miss rate. In order to explore this, the instruction miss rate is changed to 50% with the results for the issue rate shown in Figure 4.18. The system is able to continue to issue a thread every cycle as long as there are enough threads and the memory system is not congested (i.e. 128 threads with 16 eDRAMs, 128-1024 threads with 64-256 eDRAMS). However, increasing the instruction miss rate to 50% when there are only 8 threads or when the memory system is congested causes a corresponding reduction in the issue rate.

Figure 4.18. Issue Rate for increasing number of eDRAMS per LPC and Instruction Miss Rate of 50% (Eviction Limit = 0, Memory Intensive Instruction Mix)

87
All the previous results reflected a memory intensive instruction mix with 60% memory instructions. Figure 4.19 demonstrates the issue rate results for a computation intensive instruction mix with only 35% memory instructions. This less memory intensive mix reduces the pressure on the memory and increases the issue rate, requiring only 16 eDRAM per processor to support an issue rate of 1.

![Figure 4.19. Issue Rate for increasing number of eDRAMS per LPC and Computation Intensive Instruction Mix (Eviction Limit = 0, Instruction Miss Rate = 5%)](image)

Next I explore the effects of letting frames block in the frame buffer. In this
scenario a thread is allowed to block with its frame in the frame buffer unless doing so would cause the number of blocked threads in the buffer to exceed 50% (i.e. 64 frames). Figure 4.20 shows the issue rate and mean eDRAM queue lengths for one processor as the number of threads and number of eDRAM per processor are varied. This approach reduces the number of evictions and thus the memory traffic when the number of threads exceeds the size of the frame buffer. This results in higher issue rates and lower queue depths for 256 to 1024 threads per processor.

The final set of experiments varies the eviction limit to allow from 0% to 75% of the frame buffer threads to block without being evicted. Figure 4.22 shows the issue rate and mean eDRAM bandwidth for eviction limits of 0, 32, 64, and 96 with 4 processors and 4 eDRAMs. The eviction limit has no effect when the number of threads is less than or equal to the size of the frame buffer because all threads may block in the buffer. Increasing the number of frames that may block in the frame buffer from 0% to 75% reduces the memory traffic by approximately 4% and more than doubles the processor issue rate from 4% to 9% for 1024 threads.

Increasing the number of eDRAMS to 256 gives a ratio of 64 eDRAMS per processor and eliminates most of the congestion at memory. Eviction limits greater than 0 allow an instruction to be issued almost every cycle and reduce the bandwidth needed at the eDRAMs from 8 to 2.5 bytes/cycle for 1024 threads. In general, allowing more threads to block in the frame improves performance by reducing the frame swap traffic to memory.

The eviction scheme results further demonstrate the importance of the memory system in supporting processor utilization. As congestion at the memory
Figure 4.20. Issue Rate and Mean eDRAM Queue Depth for increasing number of eDRAMS per LPC (Eviction Limit = 64, Memory Intensive Instruction Mix, Instruction Miss Rate = 5%)
Figure 4.21. Issue Rate and Mean eDRAM Bandwidth for increasing eviction limit (4 Processors, 4 eDRAMs, Memory Intensive Instruction Mix, Instruction Miss Rate = 5%)
Figure 4.22. Issue Rate and Mean eDRAM Bandwidth for increasing eviction limit (4 Processors, 256 eDRAMs, Memory Intensive Instruction Mix, Instruction Miss Rate = 5%)
increases, the associated memory latencies increase and more threads become blocked. Although large numbers of threads increase the ability to hide latency, swapping threads between the frame buffer and memory adds to the congestion in memory as well. A thread policy that allows threads to block in the frame buffer improves performance by reducing the number of evictions and thus the memory traffic. Increasing the number of threads to exceed the size of the frame buffer or increasing the number of processors can both be detrimental to performance if the memory cannot handle the additional traffic. Because the experiments looked only at steady state performance, they considered only the processor and memory utilization when adding more threads or processors. This does not take into account the fact that adding more threads or processors could reduce the overall execution time. The experimental results from the non-blocking memory operations showed that performance is a balance between keeping the processor busy and limiting the congestion at memory (and thus the memory latency). Therefore, this set of experiments focused on these factors when evaluating performance effects. When extrapolating these results to a system of multiple nodes, the overall latency is expected to increase because of the remote memory references. However, the bandwidth requirements per processor should be applicable.

4.4 Conclusions

This chapter analyzed architectural tradeoffs in the baseline architecture. In particular, it evaluated the effects of memory latency, blocking and non-blocking memory accesses and thread eviction policy as well as exploring the bandwidth requirements under various architectural configurations. Two different architectural models explored system level and node level issues respectively. In the evaluation
of non-blocking memory references, performance was modeled for a system of simple nodes with each node represented as a single processor and memory. The experiments used a particular application, the vector sum, to drive both analytic and queuing simulations and used overall execution time as the primary metric. The queuing simulation additionally modeled queuing in the memory system to further quantify performance. In contrast, the evaluation of the thread eviction policy focused on a single node in greater detail. The experiments used statistically identical threads of infinite length to drive the queuing simulation and used bandwidth and issue rate (i.e. utilization) as the primary metrics. The remainder of this section discusses the key insights to be drawn from these experiments.

Experiments for evaluation of blocking and non-blocking memory references employed both analytical and queuing simulations. Results of the analytical model identified two operating regions: communication-limited and compute-limited. In the communication-limited region, there are not enough threads available to execute an instruction on every cycle and the processor must stall waiting for the results of memory accesses. In the computation-limited region, the processor is fully saturated and has enough threads to execute an instruction every cycle, eliminating the impact of latency on performance. Parallel speedup depends on the ability to mask latency and keep the processor busy. As latencies increase, the number of threads needed to saturate the processor increases as well. Supporting non-blocking memory references improves performance by reducing the number of threads needed to mask a given latency or saturate the processor. However, little additional improvement is found for allowing more than 2-4 outstanding memory references concurrently. Adding more processors has little impact on performance in the communication-limited region but improves performance when processors
are saturated. Although the analytical model provided important insight into the performance characteristics of the system, it assumed uniform latencies and was unable to model congestion in the network or at memory banks. Queuing simulation was employed to further explore these factors.

The queuing simulation results validated the results of the analytical model with one major caveat: performance is very sensitive to the memory system bandwidth, i.e. the number of simultaneous references that can be handled by the network and memory components. If the memory system can handle the memory traffic without congestion in the network or memory components, the results mimic those of the analytical model. However, congestion in the memory increases the overall latency and reduces performance. Increasing the number of threads or allowing multiple outstanding memory references both improve processor utilization but this is offset by the fact that they also increase the number of memory references issued, adding to memory congestion and thus latency.

Experiments for evaluation of the thread eviction policy employed a queuing simulation to explore the bandwidth requirements and memory configuration needed to support them. Results reinforced the importance of memory latency and bandwidth in supporting processor utilization. Large numbers of threads increase the ability to hide latency, but swapping threads between the frame buffer and memory during eviction adds to the congestion in memory. Allowing threads to block in the frame buffer improves performance by reducing the number of evictions and the associated memory traffic. Analysis concludes that the system can support 1-4 processors per LPC, with each processor requiring 16-64 eDRAMs. This corresponds to bandwidth requirements of 10-23 bytes/cycle per eDRAM and 1-4 bytes/cycle per DRAM.
Overall, the results in this chapter demonstrated the sensitivity of system performance to memory latency and bandwidth. Increasing the number of threads generally improves performance, but increased multithreading alone cannot overcome bandwidth limitations. Multiple outstanding loads reduces the number of threads needed to saturate a processor and allowing threads to block in the frame buffer reduces the memory traffic. Finally, if the processors are not highly utilized, adding additional processors provides little performance improvement.
CHAPTER 5

ANALYSIS OF MEMORY MODELS AND APPLICATION PROGRAMMING ISSUES

This chapter moves away from the architectural exploration of the previous chapter to focus on higher level issues surrounding memory addressing and multithreaded programming with the goal of understanding how applications execute on the lightweight multithreaded architecture and determining the programming and memory models best suited to the architecture and applications. Application execution on the Cray MTA-2, a multithreaded architecture, is compared to execution-driven simulation of the lightweight multithreaded architecture to explore different approaches to global addressing for the system’s distributed shared memory and evaluate the effectiveness of different multithreaded programming models including approaches to thread specification and management, distribution of work and data, and synchronization.

The remainder of this chapter is structured as follows: Section 5.1 gives an overview of the memory models and programming issues under exploration. The programming and execution models used by the LWP simulation environment and Cray’s MTA-2 are then described in Section 5.2. Section 5.3 presents a comparison of two different memory addressing models, evaluating both programming complexity and performance. Section 5.4 explores the implementation and performance on the MTA-2 of an emerging class of applications termed active graphs.
Finally, conclusions are presented in Section 5.5.

5.1 Memory Addressing Models and Programming Issues

This chapter explores the effects of data distribution in shared memory programming for the proposed architecture. In both cases the memory is modeled as a global shared address space that can be accessed via load/store semantics (as opposed to message passing). The differences arise in how data is distributed within that memory and whether the programmer has control over such placement. The term distributed global address space (DGAS) specifies a memory addressing model in which the programmer does not control data placement and virtual addresses are distributed throughout memory to reduce contention and encourage balanced workloads. Destroying spatial locality eliminates the need for the programmer to intelligently place data to achieve good performance, making it easier to program. However, it also eliminates the possibility of intelligently placing data to take advantage of locality and reduce memory access latency. In contrast, a partitioned global address space (PGAS) allows the programmer to identify whether data is local or remote. This allows the programmer to specify whether data is private to a thread, and thus should be placed in the thread’s local memory partition, or is shared among threads. PGAS also provides mechanisms that allow the programmer to relatively easily allocate data. This places the burden of data distribution on the programmer but holds the potential for improved performance through increased locality.

Two different multithreaded programming languages are explored: MTA-C and DimC. Both MTA-C and DimC languages are variants of C with extensions to support parallel execution. However, there are distinct differences in the lan-
guage support for thread specification and management, the distribution of work
and data, and synchronization. Furthermore, for a given language, different appli-
cation implementations may yield performance differences. Further details of the
programming languages are provided as a part of the discussion of the execution
models in Section 5.2.

Programming and memory models are explored through execution on the
MTA-2 and in the LWP simulation environment. MTA-C is used to program
applications for the MTA-2 which has a DGAS model where virtual addresses are
randomly distributed throughout data memory by hashing them to memory units.
Because the MTA-2’s memories and processors are interleaved on the internal net-
work, all memory requests must traverse the network to get to memory providing
uniform memory access times. The LWP simulation environment is programmed
using DimC and can be configured to represent either a DGAS or PGAS model.
For the DGAS model, data distribution mimics the MTA-2. Data is distributed
at the word level across the memory units in a round robin fashion without regard
to locality. For the PGAS model, the programmer attempts to collocate threads
with their data to optimize locality based on the data access characteristics of
the application. For both memory models in the LWP simulation environment,
the memory access time is non-uniform with local memory accesses an order of
magnitude faster than remote accesses.

Two sets of experiments explore the differences between memory models and
languages as well as between implementations within a given language. Analysis
of the results attempts to understand multithreaded application execution
and determine the programming paradigms and memory models best suited to
the architecture and applications. The first set of experiments compares the pro-
gramming complexity and performance of applications implemented in MTA-C on the MTA-2 to Sheng Li’s implementations using DimC on the LWP simulation environment for both DGAS and PGAS memory addressing models [59]. A set of four benchmarks is chosen to exhibit a range of application characteristics including shared data structures with frequent updates, barrier synchronization, and irregular dynamic execution. Analysis evaluates application performance as well as programming effort and the ability of the language to efficiently capture the application parallelism and utilize architectural features including lightweight multithreading and synchronization. The second set of experiments explores different implementations of “active graph” applications and their performance on the MTA-2. Active graphs represent an emerging class of applications with very dynamic and irregular structures and execution that limit performance on traditional architectures. An active graph model is presented that exhibits a high degree of parallelism and fine-grain synchronization to enable these applications to execute efficiently using lightweight multithreading. FFT butterfly networks are then used to explore how active graphs scale with size and connectivity on the MTA-2.

5.2 Execution Models

Two different types of execution-based analysis are employed: a custom, execution driven simulation environment and execution on the Cray MTA-2, a commercial multithreaded machine. The simulation environment allows the user to vary many of the architectural parameters as well as explore different programming mechanisms. However, simulation is limited in the size and complexity of the applications that can be explored in a timely manner and the compiler and runtime
capabilities are limited. In contrast, execution on a production machine allows the execution of larger more complex applications. In particular, it addresses one of the questions frequently encountered with multithreading: Is it possible for a compiler to extract enough threads/parallelism to effectively use the architecture? Execution on the MTA-2. demonstrates performance on a “real” multithreaded architecture, very similar to the proposed architecture, with mature compiler and runtime technology.

5.2.1 LWP Simulation Environment

The LWP simulation environment consists of the SALT simulator and DimC compiler, both developed at the Jet Propulsion Laboratory. SALT \[86\] is an execution-based simulator that models a system of lightweight multithreaded processing nodes as described in Chapter 3 including hardware parallelism, lightweight multithreading with frames, split phase memory accesses, and extension bits to support extended memory semantics. SALT is component based with multiple front- and back-ends to support efficient design modification. The compiler takes a C-like high level language called DimC \[15\] and generates MTA-style assembly source code that is interpreted by one of the SALT front-ends and then executed on the SALT simulation platform.

DimC is a multithreaded language for the LWP simulation environment. DimC takes the view that the programmer often has an in-depth understanding of the structure of an application and was developed to allow the programmer explicit control over the creation and distribution of threads and data. The DimC language is based on C and includes extensions for the explicit definition of threads, synchronization, data distribution, and the collocation of threads and data.
In DimC, the programmer uses a threaded procedure call to explicitly define a thread including its associated code, its private frame data, and its “neighborhood” of execution. The compiler creates a new thread frame for each invocation of a threaded procedure and its private data is stored in the frame. The first procedure parameter defines the “neighborhood” in which to create the thread, i.e. the thread is created and executed on the node whose associated memory contains the data indicated by the first parameter. Threads can synchronize through shared memory locations and DimC provides special syntax to indicate that an operation should use full/empty semantics and to define when the thread should block on a result. In this way, the programmer can explicitly control the creation, execution, and synchronization of threads. Unlike the MTA-2, there is no concept of hardware streams in the LWP system. Therefore, the number of threads in the system can change dynamically during execution and is limited only by the amount of memory available.

5.2.2 Cray MTA-2

The Cray MTA-2 [5, 27] is a multithreaded, shared memory architecture designed to be highly scalable, simple to program, and able to exploit the parallelism inherent in a wide variety applications. These goals were meant to address what the developers saw as deficiencies in current massively parallel computing: many programs do not have sufficient parallelism to take advantage of large numbers of processors, there is a limited body of well-written parallel code for massively parallel machines, and developing or porting parallel code can be time-consuming and difficult. Many of the ideas for the lightweight multithreaded architecture proposed here, including fine-grain multithreading and extended memory semantics,
were influenced by the MTA-2 and its predecessors [7, 85].

The MTA-2 system is organized as a collection of processors, memories, I/O caches, and I/O processors connected by a high speed interconnection network. There are 256 multithreaded (MT) processors, each consisting of an execution pipe, instruction cache, and 128 hardware streams. Each hardware stream holds the context of a thread including a program counter and general purpose registers, allowing fast context switching between threads. Since instruction interpretation is pipelined, it is possible to have an instruction issued from a different thread at each clock tick without interfering with previous instructions. Three operations can be executed simultaneously per instruction: a memory reference operation, an arithmetic operation, and a control or arithmetic operation. Explicit-dependence lookahead uses a three bit lookahead field to specify how many instructions can be issued from a given stream before encountering an instruction dependent on the current one. Together, these efforts provide parallelism and latency tolerance, reducing the need for data locality.

The MTA-2 has a uniform-access shared memory with the memories and processors interleaved on the internal network. Part of the memory near each processor is devoted to program memory to hold a copy of the instructions for each thread running on that processor. The remainder of memory is data memory and is not associated with any processor. In the data memory, virtual addresses are randomly distributed by hashing them to memory units. This reduces contention, encourages balanced workloads, and eliminates the need for careful data placement. The MTA-2 has no data caches but *hotspot* caches at each memory bank help to mitigate the performance impact of hotspots. Similar to the proposed architecture, an additional full/empty bit (as well as 3 other extension bits)
is associated with each dword. Full/empty semantics are the same in both architectures. However, the MTA-2 responds differently to the situation where a memory location is in the wrong state, e.g. full when a `store.ef` is attempting to complete. In the MTA-2, the processor retries the memory instruction until it completes. This operation is given low priority and typically waits until the processor issues an instruction where the slot allotted for memory operations is vacant. In this way, the retry consumes network bandwidth but does not delay the processor. While the thread is waiting, however, it cannot issue instructions and no other available thread can use the stream to do so. Therefore, if all 128 thread streams are waiting for memory operations to complete, the processor is stalled. There is a system option to allow a retry timeout which suspends the thread, allowing a new thread to make use of the stream. However, this option is not enabled on the system used for these experiments.

MTA-C is a multithreaded programming language for the MTA-2. One of the goals of the MTA-2 is to automatically extract the parallelism inherent in a program without the need for the programmer to explicitly identify and partition the program. Therefore, MTA-C focuses on assisting the compiler in identifying and extracting parallelism rather than explicitly defining the work and data decomposition. In order to accomplish this, the MTA-2 has a sophisticated compiler that focuses on extracting parallelism at multiple levels, such as from loops, independent blocks or functions, or at the instruction level through wide instruction words. MTA-C provides extensions, including programmer directives, futures, and synchronization variables, so that the programmer can assist the compiler in identifying parallelism.

Threads are created dynamically in MTA-C using `futures`. The future construct
indicates that a particular sequence of code can be executed as a thread, and it is best suited to implementing task-level and recursive parallelism. Loop futures provide a way of indicating to the compiler that the iterations of a loop can each be executed as a future. Although the future construct supports dynamic creation of threads, it does not provide the same level of control as DimC and requires higher overhead, limiting its effectiveness. Also, because there is no concept of local and remote memory on the MTA-2, MTA-C does not have any support for the distribution of threads or data. The programmer cannot control on which processor a thread executes or where data is placed. Therefore, there is no way to assure that a thread is located close to the data on which it works.

The MTA-2 compiler statically analyzes the code to determine the amount of available parallelism and the number of threads necessary to mask the memory latency. At runtime this information is used to determine the number of hardware streams to reserve, and work is assigned to the streams during execution. In this way, the number of threads in the system can change dynamically during execution, but the number of hardware streams available is static. The programmer can use a directive to request a particular number of streams but it is not guaranteed.

The MTA-2’s ability to tolerate latencies enables it to achieve high performance for many programs that run poorly on traditional parallel processors. The high cost of the custom system, however, limits its commercial viability. In order to address this, Cray has developed the XMT \[28, 40\] as its next generation multithreaded machine. The XMT replaces the XT3’s \[29\] processors with the MTA-2’s multithreaded (MT) processors. Although the XT3 is a distributed memory machine, the XMT treats the system as a shared memory machine. Furthermore, the XMT has less memory bandwidth per processor than the MTA-2.
5.3 Comparison of Memory Addressing Models

A set of benchmarks was chosen from *UPC: Distributed Shared Memory Programming* [37] to evaluate the memory models. UPC (Unified Parallel C) was chosen as the basis for comparison because it is the most popular language in the class of global address space languages which also includes Co-array F (CAF) [72] and Titanium [93]. Four benchmarks, matrix-vector multiplication, image histogramming, heat conduction, and NQueens, were chosen to illustrate a range of application characteristics including regular memory accesses with locality, shared data structures with frequent updates, barrier synchronization, and irregular/dynamic execution. In order to assess the differences in programming complexity and performance, each benchmark was implemented and analyzed in the LWP simulation environment for both DGAS and PGAS memory models and on the MTA-2. I implemented the benchmarks using MTA-C for the MTA-2 and Sheng Li implemented the benchmarks using DimC for the LWP simulation environment [59]. The codes were validated by doing multiple runs using a range of input values and data set sizes and then checking the correctness of the result against the results of serial implementations.

Execution time and parallel speedup are the primary metrics with the number of executing hardware streams and the issue rate (utilization) also explored for the MTA-2 to provide additional insight into performance. Direct comparison between the MTA-2 and simulation results is difficult because of differences in the way work is distributed among threads. The MTA-2 does not support the explicit distribution of threads but relies on the compiler and runtime to determine the number of threads needed to mask the memory latency, the distribution of work among those threads, and the distribution of threads within the system.
In contrast, the lightweight multithreaded architecture allows the programmer to specify the number of threads, the distribution of work, and the distribution of threads. For the MTA-2 experiments, the system size and problem size are varied. For the LWP simulation experiments, the system size, problem size, and memory model (e.g. thread and data distribution) are varied. The remainder of this section discusses the characteristics of each benchmark and the associated results.

5.3.1 Matrix Vector Multiply

Matrix vector multiply is an example of a very regular application that typically performs well on traditional architectures. It has a regular workload that can be statically partitioned and a cache-friendly memory access pattern. The vector data structure must be shared for reading but there is no synchronization required. Pseudocode for the main kernel of computation is shown in Figure 5.1.

```plaintext
for i=0 to N-1 {
    for j=0 to N-1 {
        c[i] += a[i][j] * b[j];
    }
}
```

Figure 5.1. Pseudocode for Matrix Vector Multiply
5.3.1.1 MTA-2

The DGAS memory model found in the MTA-2 simplifies program development by eliminating the need to explicitly decompose and place the data. No modifications to the sequential code were needed. The compiler was able to automatically recognize the parallelism provided by the \(i\) and \(j\) loops and optimize the reduction, \(c[i]+ = a[i][j] \times b[j]\), by pulling the store to \(c[i]\) out of the loop.

The program was executed on the MTA-2 for values of \(N\) varying from \(2^{12}\) to \(2^{16}\). Figure 5.2 shows the performance results. As \(N\) increases, performance improves because of the increased available parallelism provided by larger data sets. More parallelism yields more threads to increase processor utilization and mask latency. For \(N = 65536\) there is enough parallelism available to provide almost ideal speedup. Smaller data sets, however, do not provide the parallelism necessary to support large numbers of threads and thus large numbers of processors. For \(N = 4096\) the speedup begins to decrease for 16 or more processors because the amount of parallelism extracted is not sufficient to keep them well utilized. The lack of parallelism is reflected in the average number of streams per processor (Figure 5.2c) which drops to approximately 25 at 8 processors. The number of streams then begins to increase as the number of processors increases further but this is a result of the runtime allocating fewer iterations per thread to yield more streams. Allocating fewer iterations per thread makes more threads available to mask memory latency but also increases the overhead of thread creation relative to the total work done by the thread. Therefore, the overall issue rate continues to decrease even with additional threads available. The importance of this balance between the number of threads and the workload per thread will be seen throughout this work. Overall, the matrix vector multiply benchmark
performs well on the MTA-2 as long as the data set is large enough to provide the parallelism needed to mask the memory latencies.

5.3.1.2 DGAS Simulation

In the DGAS simulation model, each thread is assigned one or more rows of the matrix to compute depending on the number of threads and size of the matrix but without regard to locality. Unlike the MTA-2, the simulation architecture allows the programmer to specify the number of threads and their distribution. The threads are uniformly distributed among the LPCs, requiring each thread to make both local and remote memory references when accessing the matrix and vector elements.

Figure 5.3(a) shows the performance of the matrix vector multiply benchmark for $N = 8192$ with the number of threads in the system varied from 512 to 4096. As the number of threads increases, the speedup increases as well with approximately 300 threads per LPC needed to saturate the processor. For a constant number of total threads, the number of threads per LPC decreases as the number of LPCs increases. The speedup drops drastically when the number of threads per LPC falls below 64 because at that point there the number of threads is insufficient to mask the memory latency.

Figure 5.3(b) shows speedup for a constant number of threads with the data set size varied from 512 to 8192. Increasing the size of the data set increases performance slightly by giving each thread more work. However, because the number of threads is constant it does not provide any additional parallelism. These results are similar to those found with the MTA-2 and demonstrate the importance of balancing both the number of threads and the amount of work per thread.
Figure 5.2. Matrix Vector Multiply Performance on the MTA-2
5.3.1.3 PGAS Simulation

In the PGAS model simulation, the workload is distributed as shown in Figure 5.4 to optimize locality. Each thread is collocated with the rows of the \( N \times N \) matrix and result vector that it must access to perform its portion of the computation as well as one or more elements of the multiplier vector and its local data. The DimC language enables to programmer to evenly distribute the data and threads throughout the system, i.e., for \( N \) LPCs and \( M \) threads, each LPC has \( M/N \) working threads. Although the regularity of the problem makes it relatively simple to effectively distribute the work and data, programming is still more difficult than in the DGAS model which requires no such distribution.

Figure 5.5 shows the performance of the matrix vector multiply benchmark for the PGAS model. Although the trends in both DGAS and PGAS are similar, performance under the PGAS model is much better because of the reduced
memory latency provided by more local accesses. In addition, it is notable that each LPC needs approximately 100 threads to achieve ideal speedup, much less than that needed in the DGAS model. Fewer threads are needed to mask memory latency because local accesses have much shorter latency than remote accesses. For regular problems where the workload distribution is simple, PGAS provides a clear advantage.

5.3.2 Image Histogramming

Image histogramming consists of tabulating the number of pixels at each gray level for a given image. The gray levels are uniformly distributed in the range 0 to 256 requiring a histogram of size 256. This application was chosen to illustrate the architecture’s ability to handle frequent updates to shared data structures. The UPC code uses locks to ensure exclusive access when updating the histogram array. Both the MTA-2 and the LWP simulation environment use an atomic memory operation to achieve this, which improves performance by decreasing the bandwidth required.
Figure 5.5. PGAS Simulation Performance for Matrix Vector Multiply

5.3.2.1 MTA-2

No modifications to the sequential code were needed in order run on the MTA-2. The MTA-2 compiler automatically recognized the shared updates and implemented them using atomic memory operations. In addition, the hot spot caches reduced contention at the histogram array.

The program was first executed on the MTA-2 for an image of size $N \times N$ and a uniformly distributed gray level between 0 and 256. The performance results are presented in Figure 5.6. The speedup is almost ideal for up to 16 processors and then begins to flatten because it can no longer provide enough parallelism. Increasing the size of the problem increases the number of threads that can be extracted from the code. However, because the size of the histogram array does not change, increasing the number of threads also increases the contention at the shared histogram array. Overall, the latency-tolerating effects of having
more threads outweighs the additional contention at the histogram array so larger problem sizes show improved speedup.

![Graphs showing execution time and speedup](image)

(a) Execution Time  
(b) Speedup

Figure 5.6. Histogram Performance for Uniform Distribution of Image Color on the MTA-2

The MTA-2 is designed to handle one memory access per processor per cycle. This is achieved through the combination of a high performance communication network, hotspot caches, and multithreading. In order to test this ability, the histogram program was run with a range of distributions to see how “hot” the hotspot must get before performance is significantly effected. The image gray levels were distributed so that they covered a range from 0 to $R$ with $R$ varying from 256 (uniformly distributed) to 1 (single color). There were no significant differences in execution time until $R = 1$. Up to that point, the architecture
was able to effectively manage the contention at the memory. The performance of the histogram application for an image with a single color value \((R = 1)\) is presented in Figure 5.7. For all problem sizes, the execution time decreases most dramatically from 1 to 2 processors and then plateaus. As the number of processors increases the number of possible concurrent memory accesses increases as well causing increased pressure on the memory hotspot and increased delay for the atomic memory access. Although extreme, this situation provides another example in which increasing the number of threads degrades performance because the memory system cannot handle the additional accesses.

Figure 5.7. Histogram Performance for Single Image Color on the MTA-2
5.3.2.2 DGAS Simulation

In the DGAS model, each thread is assigned one or more rows of the image matrix and the data is uniformly distributed throughout the memory. Each thread requires both local and remote memory references to scan its own portion of the image array and to update the histogram array. Full/empty bit semantics are used to implement a lock to support atomic updates of the shared histogram array.

Figure 5.8 (a) shows the performance of the histogramming benchmark with a DGAS model for $N = 8192$ and different total thread numbers. Performance improves as the number of threads increases with approximately 300 threads per processor needed to tolerate the remote and local memory latencies and prevent starvation. The MTA-2 requires fewer threads and provides better speedup primarily because of the difference in memory access latencies. The MTA-2 has a custom interconnect with an average memory access latency of approximately 130 cycles. In contrast, memory access latency on the lightweight multithreaded architecture is approximately 100 cycles to local memory and 1000 cycles to remote memory\(^1\). Given enough threads, however, the architecture is able to mask its longer memory access latencies.

5.3.2.3 PGAS Simulation

In the PGAS model, each thread is assigned a portion of the image to scan as shown in Figure 5.9. A thread is collocated with its assigned rows of the image matrix and one or more elements of the histogram array. All accesses to the image are local and remote memory references are needed only to update the histogram array.

\(^1\)These values were chosen to represent the latencies in the Cascade Architecture. They were derived from estimates for the successor to the MTA-2, the Cray XMT.
Figure 5.8. Histogram Performance for DGAS and PGAS Simulations

Figure 5.9. Workload distribution for histogram in PGAS Model
Figure 5.8 (b) shows the performance of the histogram application under a PGAS model for $N = 8192$ with the number of threads varied from 512 to 4096. Approximately 100 threads are needed to keep the processor saturated and achieve linear speedup. The PGAS model provides better speedup than the DGAS model because shorter memory latencies means less threads are needed to mask those latencies. Data and thread distribution for this benchmark are relatively simple, adding little to the programming complexity. Therefore, PGAS provides clear performance advantages over DGAS.

5.3.3 Heat Conduction

This application models the transfer of heat through a three-dimensional solid over time and is representative of physics grid type applications. The heat conduction problem is structured as a three dimensional cube of size $N^3$. Computation at each point requires access to the adjacent points in the $x$, $y$, and $z$ dimensions to calculate the new temperature and the temperature change from the previous iteration. A barrier synchronization occurs between iterations and computation of the global maximum change, $dT_{max}$, over all points determines whether another iteration occurs.

5.3.3.1 MTA-2

Minimal changes were needed to enable to the program to run efficiently on the MTA-2. Programmer directives were inserted stating that the $x$, $y$, and $z$ loops had no dependencies. This allowed the compiler to parallelize across all three loops. In order to reduce the hotspot created by a single global $dT_{max}$, a set of local variables, $dT_{max}[N]$ are defined in the $x$ dimension and then a reduction is done at the end of each iteration to find the global $dT_{max}$. 

118
Figure 5.10 provides the execution time and speedup for the heat conduction problem. As the size of $N$ increases both the total problem size ($N^3$) and the number of iterations required to converge increase as well. A problem size of $N = 64$ does not provide enough parallelism for good speedup on more than 4 processors. As the number of processors increases performance degrades slightly because the total number of threads that must synchronize at each barrier increases as well. Larger problem sizes make more threads available to mask the synchronization and memory access latencies, yielding improved performance.

Figure 5.10. Heat Conduction Performance on the MTA-2
5.3.3.2 DGAS Simulation

Workload distribution was fairly easy in the DGAS model with each thread assigned one or more consecutive rows of the matrix on which to compute. However, because there is no locality, computation at each point may require up to 8 remote memory accesses to read the temperature at the current point, access the adjacent points above and below the current point in the $x$, $y$, and $z$ directions, and store the updated temperature.

Figure 5.11 (a) shows the performance in DGAS with different total numbers of threads and a problem size of $N=256$. Performance was poor for small numbers of threads. For 4096 threads, however, the ideal speedup still was obtained for up to 16 processors. These results demonstrate that lightweight multithreading is an effective solution even for those programs requiring barrier synchronization of large numbers of threads.

5.3.3.3 PGAS Simulation

Figure 5.12 shows the data distribution in the PGAS model with each thread assigned a three-dimensional sub-matrix to compute. Computations on points interior to the cube are completely local while data on the edge of the sub-matrix require the thread to access the adjacent data at the neighbors. Whether the access is remote or local depends on where the neighbor threads are located.

As expected, figure 5.11 (b) shows that the LWP can, again, achieve better speedup with a PGAS model than with a DGAS model due to increased data locality. In addition, low context switch overhead and memory-based synchronization reduces the impact of the barrier synchronization and provides the ability to efficiently support an “unlimited” number of threads. However, programming for
Figure 5.11. Speedup Comparison for Heat Conduction

Figure 5.12. Workload Distribution for Heat Conduction in PGAS
the PGAS model is significantly more difficult requiring a restructuring of the program and explicit data distribution. Rather than access data via a simple set of nested loops, the data must be divided into three dimensional submatrices with each thread computing its submatrix. Threads are cheaper than programmers, making it reasonable to consider using a DGAS model with more threads to achieve reduced performance with significantly less programming complexity.

5.3.4 NQueens

The NQueens application finds all safe placements of $N$ queens on an $N \times N$ chessboard. The main kernel of computation is the $NQueens()$ procedure that valuates a row of the chessboard based on previous placements. If a placement is valid, the procedure is recursively called to evaluate the next row, as shown in Figure 5.13. This application is representative of a broad class of problems that require the search of tree-based data structures. During a tree search it is usually impossible to determine if an entire branch will be searched or if the search will be cut off quickly. This creates a very dynamic, irregular workload that is difficult to parallelize on traditional architectures but is well suited to the dynamic multithreading provided by our architecture.

UPC provides a SPMD programming model and, as such, the original UPC implementation goes to great lengths to rewrite the recursive NQueens in a way which allows the total number of threads to be explicitly determined at compile time. In order to accomplish this the UPC code repeats some computations across threads, wasting computation. However, both the MTA-C and DimC provide constructs to support dynamic thread creation via a simple recursive implementation. In the lightweight multithreaded architecture, the compiler automatically creates
a new thread for each procedure invocation. In the MTA-2, the loop future construct allows threads to be dynamically created for each procedure invocation.

5.3.4.1 MTA-2

The MTA-2 compiler was not able to automatically recognize the parallelism found in the recursive NQueens program so the loop future directive was used to allow threads to be dynamically created for each procedure invocation. The results for problem sizes of $N = 8, 9, 10$ are shown in Figure 5.14. A problem size of $N = 8$ demonstrates superlinear speedup from 1 to 2 processors, i.e. a speedup of more than $P$ on $P$ processors. The speedup then begins to plateau as the number of processors increases and eventually drops from 16 to 32 processors. The larger problem sizes show poorer performance for small numbers of processors and better performance for larger numbers of processors.

These performance characteristics result from the way in which parallelism is extracted in the application. The NQueens problem structure is that of an N-ary
Figure 5.14. Performance of NQueens Loop Future Implementation
tree. At each level, \( L \), in the tree there are \( N^L \) possible threads, although the actual number is less because paths are cut off whenever an invalid placement is reached. So, at early levels there is little parallelism available while at lower levels the number of threads expands quickly. Normally the MTA-2 compiler estimates the number of threads needed to mask the system latencies and creates only the necessary number of threads during execution. However, using the loop future creates a thread for each recursive call regardless of whether additional threads are needed to mask latency. The overhead involved in using the future construct is much higher than the overhead involved in managing compiler extracted threads as well.

For one processor, more threads are created than are needed and the cost of creating these additional threads dominates the performance. In contrast, when executing on two processors, more of the threads can be utilized and superlinear speedup results, as seen in Figure 5.14 (b). This effect is reflected in the number of streams and issue rate as well, shown in Figure 5.14 (c) and (d). For one processor and \( N = 8 \), there are an average of 99 streams executing, but the average issue rate is only 22%. Since there is no shared data and small working sets, much of this time is being spent creating and managing threads. For two processors, the number of streams is reduced to 75 but the issue rate increases to 37%. After that point both the average number of streams per processor and average issue rate continue to decrease until at 32 processors there are only 8 streams per processor and an issue rate of 7%. As the problem size increases, performance is poorer for small numbers of processors where there are too many threads to be effectively utilized and better for large numbers of processors where more threads can be used.
The loop future directive creates many streams that can be dynamically scheduled. The overhead of creating futures is high, however, and often many more futures are created than are needed to mask latencies, increasing the overall execution times. In order to address this problem, the code was modified to restrict the number of futures that are created. An additional parameter, \( L \), was added to specify the level (row) at which additional futures should no longer be created. For example, if \( L = 2 \) then \( N^2 \) futures are created and subsequent recursive calls are handled sequentially by the existing threads. This approach provides a simple way for the programmer to manage the number of threads created to better fit the requirements of the system.

The graphs in Figure 5.15 compare the performance of the loop future implementation with levels for different values of \( L \). Figure 5.15(a) shows the execution time for a problem of size \( N = 8 \) with \( L \) ranging from 1 to 8. Execution with \( L = 2 \) gave the best performance overall with the amount of improvement inversely proportional to the number of processors. \( L = 1 \) gave the best execution time for one processor because it created the fewest threads whereas \( L = 2 \) gave the best performance when there were two or more processors. The speedup provided by creating futures only for the first two levels (\( L = 2 \)) over creating a future for each call (\( L = 8 \)) is shown in Figure 5.15 for \( N = 8, 9, 10 \). Speedups increased as the problem size increased. However, as the number of processors increase, the number of threads that can be effectively utilized increases as well. Therefore, the performance improvement provided by \( L = 2 \) becomes more limited as the number of processors increases.

In order to allow comparison with the simulation results, Figure 5.16 shows the processor speedup of the loop future implementation with \( L = 2 \) for \( N = 8, 10, \ldots \)
and 12. Speedup improves as the problem size increases but remains significantly below the ideal for more than 8 processors.

These results demonstrated that the MTA compiler and runtime can effectively parallelize many types of applications. However, for some dynamic applications the compiler is unable to effectively extract threads without explicit programmer direction. Although the future construct supports dynamic creation of threads, it does not provide the same level of control as the lightweight multithreaded architecture and requires higher overhead, limiting its effectiveness. For dynamic applications such as the NQueens, a more efficient mechanism for extracting and managing large numbers of threads could improve performance.
5.3.4.2 DGAS Simulation

The DimC language and compiler support dynamic multithreading through the use of threaded procedure calls that create a new thread for each procedure invocation. This functionality automatically parallelizes the NQueens problem by creating a new thread for each recursive call to the \texttt{NQueens()} procedure. When an invalid placement is reached during execution of the procedure, the corresponding thread dies and signals its parent thread. The low overhead thread creation, context switching and memory-based synchronization efficiently support this dynamic thread behavior.

Figure 5.17 (a) shows the speedup for 8, 10, and 12 queens for a DGAS memory model. The LWP’s threaded procedure call is much more efficient than the MTA-2’s future construct in handling the creation and management of large numbers
of dynamic threads, resulting in better speedups.

Figure 5.17. DGAS vs PGAS Speedup Comparison for NQueens

5.3.4.3 PGAS Simulation

Figure 5.17 (b) shows the speedup for the PGAS memory model. The PGAS model gives only slight performance improvements over DGAS because there is no shared data in this benchmark and each thread has a relatively small working set. The DimC language supports dynamic affinity between a thread and its data by supplying a memory pointer as a parameter when a procedure is defined. This provides an efficient mechanism to collocate each thread with its working set by placing the thread near the data specified by the pointer. Without this type of support, programming applications to take advantage of this type of dynamic
affinity would be extremely difficult.

5.4 Evaluation of Active Graphs Using FFT Butterfly Networks

High performance computing has traditionally focused primarily on regular, numerically intensive applications. However, an emerging class of high-end applications exist that are composed of graph-like structures with irregular and dynamic data and execution. These applications arise from such diverse areas as knowledge discovery, computational biology, web searching, social networks, and intelligence analysis. Furthermore, they behave very differently from typical benchmarks and in ways that challenge conventional parallel architectures. In order to address these problems, this section presents an active graph model utilizing lightweight multithreading and fine grained synchronization to enable these applications to execute more efficiently.

The remainder of this section explores the implementation and performance of active graph applications. First it explores how applications can be implemented in the active graph model using lightweight threads and fine grained synchronization. Then it investigates how active graph applications scale with size and connectivity using FFT butterfly networks. It explores the productivity and performance provided by using a high degree of parallelism and fine-grain synchronization as opposed to attempting to predetermine the execution ordering of the graph in the code.

5.4.1 Active Graph Model

An active graph model is a multithreaded programming metaphor where threads interact independently to solve a problem rather than using the traditional ap-
approach of queuing of work and sequential pointer chasing [19]. In an active graph model, an application is represented by a graph-like structure of nodes connected by edges and execution is controlled by the flow of data between nodes using producer/consumer like exchanges. The goal of such a model is to extract maximal parallelism while ensuring that each thread consumes processing resources only when it is actively processing new input.

Each node is typically implemented as a separate lightweight thread and threads communicate via shared memory locations that may utilize synchronization primitives. These memory locations may include inputs, outputs, and controls. Inputs contain the data needed for processing which is provided by upstream nodes. Outputs point to the input locations of downstream nodes to which results are written. Controls are used to coordinate execution. A thread’s operation is defined by its firing rules which specify when a node begins a new phase of computation. Some applications may require that all of a node’s inputs receive new values before beginning computation while others may begin computation as soon as any of a node’s inputs receive a new value. Different active graph structures are needed for execution control depending on the application’s execution characteristics.

When all inputs must receive new values, full/empty semantics on the input locations are sufficient to control execution. A node thread attempts to read each input location using an atomic read and empty. If the input location is empty, the thread suspends. When the location is filled, the thread is awakened, reads the new value, marks the location as empty, and moves on to the next input location. Once all input values have arrived, the thread begins processing. When processing is complete, the result is synchronously written to each of the input locations of
its downstream nodes using a store on empty. If the thread attempts to write to a location that is already full, the thread suspends. When the location is emptied, the thread is awakened, writes its value, marks the location as full, and moves on to the next location. Once the thread has finished writing its results to the input locations of its downstream nodes, the cycle begins again.

When a node may fire on any input change, each node requires an additional control location to coordinate execution. Each node thread attempts to access its own control location with an atomic read and empty. If the location is empty, then no upstream node has sent a new value, and the thread suspends. When the control location is filled, the thread is awakened, empties the location, and begins processing. Once processing is complete, the thread writes its results to the input locations of its downstream nodes as well as setting their control location. The use of an atomic memory operation to set the control location means that a node will never miss a change notification, even if it happens, for timing reasons, to handle multiple changes with a single awakening. Once the thread has finished writing its results, the cycle begins again. Other firing rules can be implemented using modifications to these basic structures.

This active graph model can be applied to many different types of applications. Production rule systems, for example, are one of the most popular knowledge representation languages used for cognitive processing. SHINE is a knowledge base tool that uses dataflow representations of the production rule system to exploit parallelism, making it well suited to an active graph model. In SHINE, knowledge bases are represented by a set of production rules that can be transformed into an acyclic dataflow graph as shown in Figure 5.18. Each node (rule) in the graph accepts inputs from some number of upstream nodes. A rule “fires”
when all of its inputs have received new values and the result is written to its downstream nodes.

Figure 5.18. Dataflow Representation of Knowledge Base

Neural network applications are also well suited to the active graph model. A neural network application is structured as a graph of neurons, where each neuron accepts numeric inputs from a some number of upstream neurons and generates outputs that go to some number of downstream neurons. Whenever one or more of the input values change, a function is applied to the current values to compute a new output value. This output value is then relayed to the downstream nodes, where the cycle repeats itself.

Each of these applications can be implemented using the active graph model
as described in [19]. Rather than focusing on a particular active graph implementation, however, this work investigates how active graphs scale with size and connectivity. Active graph structures may have many different characteristics depending on the application being explored. In order to evaluate the performance of lightweight multithreading and synchronization for a range of graph sizes and types, I synthetically generate graphs with the characteristics under exploration. These synthetic graphs and the associated implementation are based on the FFT butterfly, as shown in Figure 5.20. Due to the limited runtime and compiler support and scalability of the simulation environment, experiments consist of execution on only the MTA-2.

The FFT graph is defined by the number of input nodes, $N$, and the radix, $R$, which defines the number of downstream nodes to which a given node is connected. For a graph of radix $R$ with $N$ input nodes, there are $S = \log_R N$ stages (columns of nodes) with $N$ nodes per stage for a total of $T = S \times R + R$ nodes and $E = N \times R \times S$ edges in the graph. Graphs with different characteristics are generated for different values of $N$ and $R$, with $N$ correlating to both total graph size and degree of parallelism while $R$ specifies the number of edges incident to each node or connectivity. Because of the automated manner in which the graphs

![Neural Network](image)

Figure 5.19. Neural Network
are generated, each node executes the same code, a simple weighted sum. This does not affect the memory access and synchronization aspects of execution on which we focus. At each stage, each node in the stage reads the values from its in edges, computes the weighted sum, and writes the result to the out edges for the nodes in the next stage of the graph.

Two versions of this FFT dataflow code are developed to explore the advantages provided by the high degree of parallelism and fine-grain synchronization found in the active graph model. The first version predetermines the execution ordering of the graph, using coarse-grain parallelism to control the flow of data through the graph by executing each stage of nodes in succession. In the second version, full/empty bits are used to control the flow of data between nodes for more fine-grain synchronization and increased parallelism. Figure 5.21 provides the pseudocode for each.
for each data set
   for each stage of the graph
      #pragma mta assert parallel
      for each node in that stage
         read inputs
         compute new value
         write outputs

VERSION 1: Staged Dataflow

for each data set
   #pragma mta assert parallel
   for each node in graph
      readfe inputs
      compute new value
      writeef outputs

VERSION 2: Full/Empty Bits

Figure 5.21. Pseudocode of main loops for staged dataflow and dataflow with full/empty bits

Figure 5.22 (a) presents a small FFT graph with 4 input nodes and 3 stages to compare the thread parallelism during execution of the staged dataflow code and the full/empty bit code. The staged dataflow code controls the flow of data through the graph by executing one stage at a time, ensuring that each node has the correct data available from the previous stage when it begins execution. At each stage, a thread is created for each node in the stage and these threads run in parallel, as shown in Figure 5.22 (b). Each node thread reads its input data, computes the weighted sum, and then writes the result to its out edges. A barrier is performed to wait until all threads in the stage have completed before starting the next stage. This allows all nodes within a stage to execute in parallel but prevents a node in the next stage from beginning execution until ALL nodes in the previous stage have completed.

In contrast, the full/empty bit dataflow code uses fine-grained synchronization to control the flow of data through the system and all nodes in the graph are allowed to execute in parallel, as shown in Figure 5.22 (c). A thread is created for
every node in the graph. Each node thread tries to read its input data. If it finds an input to be empty it blocks, waiting for the data to arrive. Once all the input data arrives, the node computes the weighted sum and then does a synchronized write to the out edges, blocking if it finds an edge still full. This allows a node to begin processing as soon as it has its input values available as opposed to having to wait for all nodes in the previous stage to complete. Comparing these two implementations identifies the characteristics of active graph applications that make them best suited to the proposed architecture’s lightweight multithreading and the fine-grain synchronization provided by full/empty bits.

5.4.2 Results

The experiments were run on the MTA-2 using 10 processors. FFT graphs were generated with 1024 to 65536 input nodes and both radix 2 and radix 4 as
The first set of experiments explored the performance characteristics for radix-2 FFT butterfly dataflow graphs of increasing size. The graphs were generated by doubling the number of inputs at each data point which approximately doubles the total number of nodes in the graph. Inputs sizes ranged from 1024 to 65536.
nodes corresponding to 11,264 to 1,114,112 total nodes in the graph as given in Table 5.1. Figure 5.23 shows the performance characteristics associated with these graphs for both the staged dataflow and the full/empty bit implementations of the application.

We would expect the execution time to double as the size of the graph doubles. However Table 5.2 demonstrates that, for the smaller graphs, execution time increases only minimally as the graph size increases. For example, increasing the graph from 11264 to 106496 total nodes increases the execution time for the staged dataflow from 0.242 to 0.364 seconds while the full/empty bit version increases from 0.141 to 0.263 seconds corresponding to less than a 2X increase in execution time for almost a 10X increase in graph size. This occurs because the smaller graphs do not have enough parallelism to saturate the processor and mask the memory latencies. Therefore, as the graph size increases the processor is able to execute more efficiently. This effect is reflected in both the average number of streams per processor and average issue rate per processor presented in Figures 5.23(b) and (c) respectively. As the number of streams increases from 88 to 100 streams, there is a corresponding increase in issue rate from 0.2 to 0.75. So, as the size of the graph increases, more parallelism is gained and execution is interleaved to more fully utilize the processor. Once graph sizes of approximately 500,000 nodes are reached there is a corresponding plateau in both the number of streams and the issue rate. At this point the processor begins to saturate with over 100 streams and reaches approximately 85% utilization and the expected doubling of execution time with doubled graph sizes occurs. So, increasing the size of the graph has little effect on execution time until the system approaches saturation.

In addition to exploring how multithreaded active graphs problems scale, we


### TABLE 5.2

**EXECUTION TIME FOR FFT RADIX-2 GRAPHS**

<table>
<thead>
<tr>
<th>N</th>
<th>Full/Empty (sec)</th>
<th>Staged (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11264</td>
<td>0.141</td>
<td>0.242</td>
</tr>
<tr>
<td>24576</td>
<td>0.157</td>
<td>0.272</td>
</tr>
<tr>
<td>53248</td>
<td>0.194</td>
<td>0.31</td>
</tr>
<tr>
<td>106496</td>
<td>0.263</td>
<td>0.364</td>
</tr>
<tr>
<td>245760</td>
<td>0.445</td>
<td>0.53</td>
</tr>
<tr>
<td>491520</td>
<td>0.722</td>
<td>0.75</td>
</tr>
<tr>
<td>1114112</td>
<td>1.531</td>
<td>1.44</td>
</tr>
</tbody>
</table>
Figure 5.23. Performance characteristics for staged dataflow and dataflow with full/empty bits on radix-2 FFT butterfly graphs
investigate the use of full/empty bits in programming such problems. Overall the performance characteristics of the two versions are very similar. For smaller graphs the full/empty bit version provides better performance; as the graphs increase in size the staged dataflow version begins to perform slightly better. This is due to two factors: the overhead involved in going parallel and the amount of parallelism available. For smaller graphs the full/empty bits provide an advantage because they provide lower overhead for going parallel, once for all nodes rather than once for each stage of nodes. The full/empty bits also allow nodes from later stages to begin execution sooner. Thus, when the graphs are smaller and the amount of available parallelism per stage is less, the combination of increased parallelism and lower overhead provided by the full/empty bits results in better performance. However, for larger graphs the number of nodes in each stage, and thus available parallelism, has grown to the point that the staged version almost fully saturates the processor and the full/empty bits no longer provide that advantage.

The second set of experiments explored the effects of increasing the number of edges in the graph by generating FFT butterfly graphs of radix 4. This increases the number of edges per node from 2 to 4 but it also has the effect of increasing the number of nodes per stage, and thus the available parallelism, as compared to radix-2 graphs of approximately the same total size. If we consider our dataflow algorithm, doubling the number of edges between nodes doubles both the number of inputs and the number of outputs. For each node this increases the memory operations by a factor of 4 but increases the computation per node by only a factor of 2 thereby increasing the number of streams needed to mask the memory latency. As a consequence, our performance results show that the increase in parallelism seems to approximately balance the increase in memory references.
and their associated latency, leading to performance very similar to that of the radix-2 examples.

Figure 5.24 gives the performance characteristics for the radix-4 FFT dataflow experiments. The graphs were generated with inputs of 1024 to 65536 nodes corresponding to total graph sizes of 6144 to 589,824 nodes. Figure 5.24 (a) presents execution times very close to that of the radix-2 experiments even though the number of edges at each node has doubled. The average streams (b) and issue rate (c) are also very close to those of the radix-2 graphs. However, a significant difference is seen in the number of memory references, presented in subfigures (d). For the radix-4 example, the average memory references per processor per cycle peaks at approximately 0.45 for 600,000 nodes whereas the radix-2 graphs peak at approximately 0.4. As the number of edges per node increases, the number of memory references increases as well. Therefore, active graphs with larger numbers of edges need an associated increase in arithmetic operations and/or parallelism to offset the increased memory latency.

Comparing the performance of the staged dataflow and full/empty bit dataflow on the radix-4 graphs shows similar trends to those found with the radix-2 graphs. Increasing the size of the graph from 6144 to 28763 total nodes gives a corresponding change in execution time of only 0.199 to 0.236 seconds for the staged dataflow code and 0.137 to 0.175 seconds for the full/empty bit dataflow code. However, increasing the graph size from 131072 to 589824 total nodes provides enough parallelism to more fully saturate the processor and corresponds to an increase in execution time from 0.395 to 1.017 for the staged dataflow and from 0.359 to 1.14 for the full/empty bit dataflow. Again we see that for small graphs the staged dataflow cannot provide enough parallelism and the full/empty bit version pro-
Figure 5.24. Performance characteristics for staged dataflow and dataflow with full/empty bits on radix-4 FFT butterfly graphs
vides better performance. Large graphs provide enough parallelism in both the staged and full/empty bit dataflow codes to begin to saturate the processor and, at this point, they begin to provide similar performance.

5.5 Conclusions

This chapter evaluated the high level issues surrounding memory addressing and programming for the proposed architecture using execution-based analysis. The first set of experiments evaluated the programming complexity and performance of multithreaded applications under both DGAS and PGAS memory addressing models. The second set of experiments then explored various multithreaded implementations of active graph applications on the MTA-2 and how they scale with size and connectivity. Together the results are used to understand how the memory model and programming paradigms support multithreaded application development and performance.

Results showed that both the MTA-2 and lightweight multithreaded architecture exhibit good parallel speedup for a variety of applications with a range of execution characteristics including frequent data sharing, hot spots, barrier synchronization, and an irregular/dynamic workload. The MTA-2 provided an established architecture with mature compiler and runtime technology on which to explore multithreaded performance. Although direct comparison between the MTA-2 and LWP simulation results is not possible because of differences in the distribution of work and data, the performance trends are very similar. The MTA-2 results demonstrated the compiler’s ability to automatically extract large amounts of parallelism with little programmer intervention and indicated that the MTA-2 typically requires fewer threads than the proposed architecture for similar
speedups due to the lower memory access latency. Furthermore, for the same problem size, the MTA-2 frequently gave better speedup than the PGAS model for the largest number of threads. This is due, in part, to the smaller memory latencies but is probably also due to the advanced compiler and runtime abilities to extract additional parallelism and provide optimizations. If the systems had comparable memory latencies (as well as a comparable compiler and runtime), I would expect the PGAS implementation on the LWP to give the best performance. For applications with locality, it has an obvious advantage in its ability to distribute work and data. For applications with large amounts of dynamic parallelism the lightweight multithreaded architecture’s lower overhead thread management provides an advantage, as shown in Figure 5.25. Figure 5.25 plots the speedup on the MTA-2 and LWP architecture for the dynamic NQueens benchmark. The LWP architecture’s hardware control for dynamic multithreading provides much better performance than the software-based future used on the MTA-2. The PGAS implementation provides additional speedup by allowing the threads to execute “close” to the data on which they compute.

Results of both the MTA-2 and LWP execution validated the multithreaded performance trends found in the previous chapter using statistical workloads. Increasing the number of threads improves performance until the processors begin to reach saturation. Additional threads after that point, however, do not provide significant performance improvements. Conversely, increasing the number of processors does not provide significant performance improvement unless there are enough threads to saturate the processors.

Analysis identified key differences between the programming languages and execution models for the two platforms. DimC provides more explicit control over
the creation and execution of threads as well as the ability to distribute data and threads to take advantage of affinity while MTA-C relies more on identifying opportunities for parallelism, with the compiler and runtime responsible for defining thread execution. On the lightweight multithreaded architecture, the number of threads changes dynamically during execution and is limited only by the available memory. On the MTA-2, the compiler analyzes the code to determine the number of threads to create and reserves the streams at runtime. The number of available streams remains constant throughout execution and is limited to 128 per processor. Both the MTA-C and DimC programming languages have features to support the explicit creation of threads during runtime. DimC’s threaded procedure call is more intuitive and provides more control than MTA-C’s future directive. Also, DimC’s lightweight thread frame representation and efficient thread blocking provide lower overhead for dynamic threading than that involved in executing futures
on the MTA-2. Therefore, DimC on the proposed architecture is better suited to supporting applications with large numbers of dynamic threads.

Experiments demonstrated that programmers can improve performance in a multithreaded architecture in two ways: by exploiting locality and hence reducing latency, or by increasing the available parallelism. The PGAS memory model allows the programmer to reduce the average memory access latency by distributing the workload and data to take advantage of locality. For those applications that exhibit a significant degree of locality, programming the application using a PGAS model requires 1/2 to 1/3 as many threads to saturate the processor as using a DGAS model with no locality for the lightweight multithreaded architecture. Therefore for the same number of threads, using the PGAS model provides better performance than the DGAS model. However, the performance improvement provided by the PGAS model depends on the programmer’s ability to decompose the work and data effectively to reduce remote accesses and may greatly increase the programming complexity required to achieve good performance. For problems with minimal locality or locality that is difficult to extract, DGAS provides a simpler programming model. Furthermore, in some cases this simpler model may provide equivalent or better performance by allowing the programmer to focus on increasing the available parallelism rather than increasing locality. The choice of memory model depends on both the application characteristics and the available time for application development but, in general, the PGAS model provides better performance.

Experiments also explored the implementation and scaling of active graph applications. The active graph model provides an efficient way to program applications for the proposed system by taking advantage of the architecture’s support
for large numbers of threads, dynamic parallelism, and low overhead synchronization. As the number of nodes in an active graph increases, the number of threads increases as well. This improves performance until the processors begin to saturate and then the overhead of creating and managing threads may cause subsequent performance to drop. As the connectivity increases, the number of communications between threads and thus the number of memory accesses and synchronizations increase. This requires an accompanying increase in either arithmetic operations or the number of parallel threads to offset the increased latency.

For active graphs that can be efficiently scheduled in the code, there is some performance benefit to doing so. However, for many graphs it is either impossible or time prohibitive to determine a scheduling of loops in the code that will provide sufficient parallelism. The code may be highly irregular, difficult to stage, or the available staging may not provide enough parallelism. In these cases the use of full/empty bits allows the programmer to parallelize over all nodes, spending much less time in analyzing and tuning the code for equivalent, if not improved, performance. In general, static scheduling provides performance improvements over dynamic scheduling for those applications where the parallel execution can be easily understood and extracted. Dynamic parallelism with large numbers of synchronizing threads is an effective way to extract parallelism for those applications where it is difficult or impossible to explicitly define the parallelism a priori. Although only one set of experiments is shown for the active graph applications, many of these trends were also seen for other dynamic applications including the NQueens applications discussed previously and the satisfiability solver presented in the next chapter.
CHAPTER 6

EVALUATION OF A MASSIVELY MULTITHREADED ARCHITECTURE
FOR SATISFIABILITY PROBLEMS

In previous chapters, statistical and execution based methods were used to explore the performance characteristics of a massively multithreaded architecture for a variety of benchmarks. These explorations provided an understanding of the sensitivity of performance to key system and application level characteristics. This chapter attempts to synthesize this information by evaluating the proposed architecture in the context of a specific application, a multithreaded solver for satisfiability problems. The satisfiability problem is an important problem with widespread applicability and embodies many characteristics that make it difficult to parallelize on traditional architectures including an irregular, dynamic workload requiring thread synchronization and data sharing. This work explores the characteristics of this application and its performance on the MTA-2 as well as in the SALT simulation environment. This information is then combined with earlier results to describe the key characteristics of the system architecture needed for optimum performance.

Section 6.1 discusses the satisfiability problem and its characteristics. Sequential and parallel solvers are discussed in Section 6.2. The benchmarks and the multithreaded solver implementation used for evaluation are then described in Sections 6.3 and 6.4. The results of execution on the MTA-2 and in the SALT
simulation environment are presented in Section 6.5. Section 6.6 combines this information with the results of previous experiments to discuss the characteristics of a massively multithreaded architecture needed to support performance for the satisfiability problem. Finally, conclusions are presented in Section 6.7.

6.1 The Boolean Satisfiability (SAT) Problem

The Boolean Satisfiability Problem (SAT) poses the question: Given a boolean formula is there some way to assign TRUE and FALSE values to the variables to make the entire expression TRUE? Solving the SAT problem has been of great interest to researchers because of its wide practical applicability and was the first problem proven to be NP-complete \[26\]. Many problems from a variety of disciplines can be encoded as SAT problems including circuit design \[61\], circuit testing \[87\], logic synthesis, \[55\], FPGA layout \[67\], scheduling \[13\], software verification \[47\], and AI planning \[50\]. SAT applications typically have very dynamic workloads that are difficult to parallelize on traditional architectures but make it well suited to a lightweight, multithreaded architecture.

Solving a SAT problem consists of finding a satisfying solution to a boolean formula or determining that no such solution exists. If such an assignment exists the formula is termed satisfiable, if not the formula is unsatisfiable. SAT problems are typically represented as a formula in conjunctive normal form (CNF) over a set of Boolean variables \((x_1, x_2, ..., x_n)\). In conjunctive normal form, a literal, \(l\), is a variable \(x_i\) or its negation \(\bar{x}_i\). A clause, \(c\), is a logical OR of literals, \((l_i \lor ... \lor l_j)\). A formula, \(f\), is a logical AND of clauses, \((c_i \land ... \land c_j)\).

Conjunctive normal form is used because it provides simplifying characteristics that aid in solving the problem and any boolean equation can be transformed into
an equivalent equation in CNF format. If one literal in a CNF clause is TRUE then the entire clause evaluates to TRUE regardless of the values of the other literals. This characteristic forms the basis of Boolean Constraint Propagation (BCP), the process of searching for unit clauses and their associated implications. In a unit clause, one literal is unset and all other literals have the value FALSE. In order for a unit clause to evaluate to TRUE, this unset literal must be set to TRUE. This assignment is termed an implication. In this way, setting a variable’s value may lead to many other variables being assigned a value through implication. In order for any CNF formula to evaluate to TRUE, each individual clause in the formula must evaluate to TRUE. This characteristic allows clauses to be evaluated individually. If any clause evaluates to FALSE then the entire formula is FALSE, and evaluation of the remaining clauses can be avoided for this set of variable assignments. If all clauses evaluate to TRUE then the entire formula is satisfied.

Solving SAT problems can be computationally complex both in time to solution and memory requirements. Because each variable in the formula can be assigned either a TRUE or FALSE value, a problem with $N$ variables has $2^N$ possible assignments. This can be envisioned as searching a binary tree where each variable is assigned either a T or F value (Figure 6.1). Practical problems can range in size from hundreds of variables and clauses to millions. Depending on both the problem and search method, a satisfying solution may be found with few assignments or, for an unsatisfiable problem, the entire tree may need to be searched. Due to the possibly long execution times, the inability to reliably determine which problems will be solved quickly, and large memory requirements research into faster, more efficient SAT solvers is ongoing.
6.2 SAT Solvers

6.2.1 Sequential Solvers

Over the years researchers have developed many different SAT solvers but the most widely used solvers today fall into two main classes: complete and incomplete solvers. Complete solvers typically use the Davis-Putnam (DP) backtrack search method [34] to enumerate and examine the possible variable assignments. They are guaranteed to find a solution if the problem is satisfiable or to terminate once all possible variable assignments have been explored if the problem is unsatisfiable. SATO [94], Chaff [65], Grasp [63], and Satz [58] are examples of complete solvers. Incomplete solvers typically use local search methods to find a solution more quickly than complete solvers. Incomplete solvers, however, are not guaranteed to find a solution, particularly if a problem is unsatisfiable. Walksat [82] and G-SAT [81] are examples of incomplete solvers. Because of the computational
complexity of solving SAT problems, research continues into the development of algorithms to solve the problem faster as well as methods to solve larger, more difficult problems.

The DP algorithm is the basis for most successful complete solvers. It enumerates the assignment of variable values in a way that is likely to yield a solution quickly. In its most general form it consists of assigning a value to a variable (decision) and then using BCP to assign other variable values implied by the original assignment (implication). If, during implication, a contradiction is discovered, then the most recent decision value and its implications are “undone” (backtrack) and the opposite value assigned. If both values have been assigned then the algorithm must backtrack further up the tree until a decision point is reached that has not been tried both ways. The three main phases involved in the DP algorithm are described in more detail below.

1. **Decision.** Choose a variable out of the set of unassigned variables and assign to it a value of TRUE or FALSE. This assignment is termed a decision point and a decision stack is used to keep track of each decision point and its associated state.

2. **Implication.** When a variable has been assigned a value, the current set of clauses is searched for unit clauses. For each unit clause encountered, the associated variable assignment is applied which makes the clause TRUE. This process continues until all clauses evaluate to TRUE, there are no unit clauses, or a contradiction is found. If all clauses are TRUE then execution halts and the current (possibly partial) set of variable assignments is a satisfying solution. If there are no unit clauses remaining then execution returns to phase 1 for a new decision. If a contradiction is found then
execution must backtrack. A contradiction occurs when two different unit clauses imply opposite values for the same literal. Note that for a given decision there may be many associated implications, each of which may trigger additional implications of its own.

3. **Backtrack.** When a contradiction is found then the current set of partial assignments cannot satisfy the formula and must be undone to some previous point. The current (most recent) decision point on the stack is located and the decision assignment and all associated implications are “undone”. The decision is then “redone” by assigning the opposite value to the variable and continuing on to implication. If both values (T and F) have been tried for the decision variable then backtrack continues further down the stack until a decision point is reached for which both values have not yet been tried. If no such decision point exists then execution halts and the problem instance is unsatisfiable.

Using this approach all $2^n$ variables may be searched in the worst case. However, in practice implications and contradictions found in phase 2 often drastically reduce the search space. Numerous optimizations have also been developed to make this search even more efficient. Most solvers today use some type of decision strategies and/or learning. Decision strategies are methods for choosing the decision variable and its value. These strategies range in complexity from randomly choosing a variable to complicated heuristics based on the current state of variables, literals, and clauses. For a discussion of decision strategies see [62]. Learning encompasses methods for utilizing the information gleaned from contradictions to prune the search space. Contradictions are analyzed to determine what partial variable assignment caused them. This information may be used
to generate conflict clauses that are added to the clause database so that these assignments are avoided in the future or to determine to which decision point to return during the backtrack phase. Each algorithm has its own set of heuristics that it uses to optimize performance.

Two simplified examples of using the DP algorithm are presented in Table 6.1. In the first example there are five clauses each consisting of two or three literals. In the decision step, variable \( x_1 \) is assigned a value of \( T \). After this assignment, clause \( c_1 \) is satisfied and clause \( c_3 \) becomes a unit clause with the implication that \( x_4 \) must be set to \( T \). This assignment satisfies clause \( c_3 \) and \( c_4 \) becomes a unit clause with the implication that \( x_2 \) must be set to \( F \). Clause \( c_4 \) is then satisfied and \( c_2 \) becomes a unit clause with the implication that \( x_3 \) must be assigned \( T \). After \( x_3 \) is assigned there are no unit clauses remaining so execution returns to the decision step where \( x_5 \) is assigned the value \( T \). This satisfies the final clause and thus the entire formula. Note that in this partial assignment the value of \( x_6 \) does not matter.

In Example 2, the initial decision step assigns the value \( T \) to variable \( x_1 \). This satisfies clause \( c_1 \) and makes both \( c_3 \) and \( c_5 \) unit clauses. However, these two unit clause present a contradiction because clause \( c_3 \) implies \( x_4 \) should be set to \( T \) while \( c_5 \) implies it should be set to \( F \). Execution must then backtrack to the previous decision and change the assignment of \( x_1 \) to \( F \). Since there are no unit clauses after this assignment, another decision is made and \( x_3 \) is assigned \( T \). Again, no unit clauses remain, so \( x_2 \) is assigned the value of \( F \), satisfying the last clause.

In both examples the choice of decision point and value were made rather arbitrarily. For any particular algorithm, the decision strategy used could change the order in which variables are assigned and/or the initial value assigned. The
### TABLE 6.1

**EXAMPLE APPLICATION OF DP TO SAT PROBLEMS**

<table>
<thead>
<tr>
<th>Example 1</th>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c_3$</th>
<th>$c_4$</th>
<th>$c_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 \lor x_3 \lor \bar{x}_5$</td>
<td>$x_2 \lor x_3 \lor \bar{x}_4$</td>
<td>$\bar{x}_1 \lor x_4$</td>
<td>$\bar{x}_2 \lor \bar{x}_4$</td>
<td>$x_5 \lor \bar{x}_6$</td>
<td></td>
</tr>
<tr>
<td><strong>Decision:</strong> $x_1 = T$</td>
<td>sat</td>
<td>$x_2 \lor x_3 \lor \bar{x}_4$</td>
<td>$F \lor x_4$</td>
<td>$\bar{x}_2 \lor \bar{x}_4$</td>
<td>$x_5 \lor \bar{x}_6$</td>
</tr>
<tr>
<td><strong>Implication:</strong> $x_4 = T$</td>
<td>$x_2 \lor x_3 \lor F$</td>
<td>sat</td>
<td>$\bar{x}_2 \lor F$</td>
<td>$x_5 \lor \bar{x}_6$</td>
<td></td>
</tr>
<tr>
<td><strong>Implication:</strong> $x_2 = F$</td>
<td>$F \lor x_3 \lor F$</td>
<td>sat</td>
<td>$x_5 \lor \bar{x}_6$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Implication:</strong> $x_3 = T$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Decision:</strong> $x_5 = T$</td>
<td>sat</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
</tr>
<tr>
<td>$x_1 \lor x_3 \lor \bar{x}_5$</td>
</tr>
<tr>
<td><strong>Decision:</strong> $x_1 = T$</td>
</tr>
<tr>
<td><strong>Contradiction:</strong></td>
</tr>
<tr>
<td>$x_4$ in $c_3$ and $c_5$</td>
</tr>
<tr>
<td><strong>Backtrack:</strong> $x_1 = F$</td>
</tr>
<tr>
<td><strong>Decision:</strong> $x_3 = T$</td>
</tr>
<tr>
<td><strong>Decision:</strong> $x_2 = F$</td>
</tr>
</tbody>
</table>
addition of learning strategies would also change the search path. Thus, for the same problem, different implementations may reach different solutions or the same solution by a different path.

6.2.2 Parallel Solvers

Although many improvements have been made in the performance of SAT solvers, there are still many problems for which sequential solvers take too long to complete or cannot complete because of memory constraints. Therefore, research has turned to the development of parallel SAT solvers. Parallelization of SAT solvers is difficult for a number of different reasons but one of the greatest impediments to parallel implementations is the dynamic nature of the workload. It is impossible to know a priori how to divide the search space for uniform distribution of the workload. Also, many algorithms use heuristics which necessitate the sharing of large data structures in parallel implementations.

Most parallel implementations use some form of message passing on distributed machines. Parallel Satz [49] and PSATO [95] divide the search space into independent subspaces that can be explored without the sharing of information between parallel tasks. These implementations are targeted to clusters of networked machines with distributed memory. Boehm and Speckenmeyer [16] present an implementation without sharing for a message-based MIMD machine while Gridsat [24] implements a parallel solver with the exchange of learned clauses for a widely distributed, heterogeneous grid. More recently, multithreaded implementations have been developed but their success has been widely varied. PaSAT [14, 83], ySAT [38, 39], and MiraXT [57] are all multithreaded solvers that include the exchange of learned clauses. For both PaSAT and ySAT, performance degradation was
seen as the number of threads per processor increased. In ySAT this effect was attributed to cache interference while for PaSat the additional synchronization overhead was blamed. MiraXT, on the other hand, performs well for two threads per processor. Its superior performance is due, in part, to an efficient implementation of the shared clause database. All three of these solvers use heavy-weight threads.

The lightweight multithreaded architecture effectively addresses many of the issues that restricted performance in these other multithreaded applications. The shared memory system without a traditional data cache eliminates cache interference. Support for low-overhead, lightweight multithreading and efficient synchronization reduces the cost involved in having multiple threads executing concurrently. In addition, extended memory semantics provides efficient support for sharing of information between threads.

6.3 SAT Benchmarks

Due to the nature of SAT problems, execution of solvers on different benchmarks can be non-deterministic, making it difficult to quantify performance. Times may vary significantly between runs of a single benchmark due to both the variability of the search and the non-determinism involved in parallel multithreading. Satisfiable benchmarks typically run faster than unsatisfiable benchmarks of equivalent size because less of the solution space must be explored. However, they have more variability in execution across runs and may show superlinear speedup with parallel implementations. This occurs because changing the order of the search may enable them to find the same solution, or an entirely different solution, earlier in the search. In contrast, unsatisfiable problems make it easier to compare
performance because the entire space must be searched but they run longer and require more memory. A variety of both satisfiable and unsatisfiable benchmarks were chosen from the SATLIB website \[44\] to represent a range of problem sizes and difficulties.

**Uniform Random 3-SAT**: These benchmarks (uf* and uuf*) consist of randomly generated clauses with 3 literals. Given \(N\) variables there are \(2^N\) possible literal values consisting of the variables and their negations. Clauses are generated by randomly choosing literals and a clause is discarded if it contains the same literal more than once or contains a literal and its negation. These problems are considered hard problems because of a particular property of uniform random 3-SAT called the *phase transition phenomenon* \[23\]. For a fixed problem size of \(N\), the probability that the problem is satisfiable drops sharply when the number of clauses reaches a critical point \(K'\). For small \(K\), there are fewer constraints and the problem is likely to have a solution. For larger \(K\), the problem is likely over constrained and unsatisfiable. For random 3-SAT, this phase transition occurs at approximately \(\alpha = k'/n = 4.26\). Analysis shows that problems from the phase transition regions tend to be particularly hard. The values of \(n\) and \(k\) in these instances were chosen to fall into this phase transition region making these difficult problems.

**Blocks World**: Blocks World is a well known problem in AI planning. It aims to find a sequence of moves that takes a set of blocks from an initial configuration to a final goal configuration.

**All-Interval Series**: The All-Interval Series (AIS) problem is inspired by a well-known problem occurring in serial musical composition and aims to find a series of \(n\) numbers with specific constraints on the intervals between them. This
type of problem can be very hard for local search methods but reasonably easy for systematic search methods.

6.4 A Multithreaded SAT Solver Implementation

A multithreaded architecture with low overhead threads is a good fit for a parallel SAT solver’s dynamic workload. In order to explore the architectural characteristics best suited to solving the SAT problem, a multithreaded SAT solver derived from the Chaff algorithm [65] was implemented. The Chaff algorithm was chosen because it forms the basis for one of the fastest and most popular sequential solvers, zChaff [96]. Chaff incorporates a number of optimizations including the VSIDS decision strategy and conflict resolution. However, as noted by Moskewicz in [65] approximately 90% of the solver’s execution time is spent in the BCP search, making an efficient BCP engine the key component. Therefore, our solver implements the core BCP engine without complicated optimizations. Once this core implementation has been explored, decision strategies and learning can be added to further improve performance. The multithreaded implementation described here is derived from a sequential solver implementation called sew developed by Kogge [51].

The chaff algorithm implements BCP using the concept of watched literals. Watched literals is a technique that reduces the number of clauses that must be evaluated as a part of BCP when a variable assignment is made. Instead of checking every clause every time an assignment is made, the only clauses that need to be evaluated are those in which a watched literal refers to the assigned variable. For each variable, a linked list of such clauses, a watchlist, is kept. We use a single-ended watchlist in which the following properties are maintained for
each clause:

- one literal is marked as “watched”

- all literals to the left of the watched literal evaluate to FALSE

- all literals to the right of the watched literal have not yet been evaluated

- the clause is linked onto the watchlist of the variable to which its watched literal refers.

Two types of list data structures are used in this approach. A variable evaluation list keeps track of all variables that have been assigned values and, for each variable, an associated watchlist keeps track of those clauses that need to be evaluated with the new assignment. Initially, the leftmost literal in each clause is considered watched and the clause is linked onto the appropriate watchlist. Once a variable has been assigned a value then the clauses on its watchlist can be evaluated. Evaluation starts with a clause’s watched literal and, moving to the right, literals are examined until either a literal is found that does not evaluate to FALSE (i.e. evaluates to TRUE or is unassigned) or the last literal in the clause has been examined (i.e. all literals in the clause evaluated to FALSE). If the literal evaluates to TRUE then the clause is marked as satisfied. If the literal is unassigned and it is NOT the last literal in the clause then this literal becomes the watched literal and the clause is linked onto the associated variable’s watchlist. If the literal is unassigned and it IS the last literal in the clause then it is a unit clause. The associated variable is assigned the value that makes the clause TRUE, the clause is marked as satisfied, and the variable is linked onto the variable evaluation list. If all literals in the clause have been examined and evaluate to FALSE then the entire clause is FALSE and a contradiction has been
found. This process is applied to each clause on a variable’s watchlist. Figure 6.2 presents pseudocode generalizing the core of the program.

```c
Solve() {
    while (1) {
        BCP()
        if satisfied then done
        if contradiction then Backtrack()
        else Decide()
    }
}

BCP() {
    for each variable on the variable evaluation list
        for each clause on the variable’s watchlist
            Evaluate_clause()
}
```

Figure 6.2. Pseudocode for core of SAT program

Parallelism can be found at a number of different levels with this algorithm. For coarse-grain parallelism, implementations focus on splitting the search space at the decision level. For each variable, an assignment of either TRUE or FALSE can be made representing two different branches of the search space. In the sequential algorithm, the first branch is tried and then, if a backtrack occurs, the other branch is explored. In a parallel implementation the branches may be explored concurrently. At a more fine-grain level, parallelism may be found in the loops of the BCP procedure by evaluating different variables on the variable evaluation list concurrently or by evaluating the clauses on a given variable’s
watchlist concurrently. I developed a multithreaded implementation to extract parallelism at all three of these levels. Because of the characteristics of the MTA-2, however, only the coarse-grain parallelism could be used effectively.

Since the MTA-2 was the initial target machine, loop level parallelism was explored first. Because the MTA-2 is very effective at extracting loop level parallelism, this approach required minimal changes from the sequential implementation. The variable evaluation list and watchlists were changed from a pointer-based list to an array-based list so that the MTA-2 could recognize and extract the loop-level parallelism. Also, operations such as assigning a value to a variable, adding to a list, and removing from a list were made atomic using extended memory semantics to avoid conflicts during parallel evaluation of the clauses.

This approach, however, did not provide the expected performance for two reasons: the overhead involved in going parallel was larger than expected and the size of the watchlists and variable evaluation list were smaller than expected. In order to support parallel execution, the linked lists were changed to arrays. Because copies of these lists are made at each decision point, the cost of creating and copying these arrays was prohibitive. Analysis of the characteristics of the benchmarks revealed that the variable evaluation list and watchlists did not provide the expected parallelism. For a problem with $N$ variables and $M$ clauses, an average of $\alpha = M/N$ clauses refer to a given variable (assuming a variable appears only once per clause), giving an approximation of the watchlist size. Table 6.2 summarizes these benchmark characteristics. The MTA-2 needs tens to hundreds of threads per processor to balance the overhead involved in creating threads and sharing data. However, for the benchmarks explored, neither the watchlists nor the variable evaluation lists provide sufficient parallelism. Although some of the
benchmarks had large numbers of variables, the size of the variable evaluation lists during execution is very dynamic and difficult for the MTA-2 to parallelize.

Splitting at the decision level is the approach used by most current parallel implementations. For each decision point in the multithreaded implementation, the original thread follows one branch of the search tree and a child thread can be created to follow the second branch. Each thread runs independently with its own copy of a subset of data structures. The formula array is shared but each thread maintains its own decision variable, variable data, and clause data. When a thread finds a satisfying solution, it saves the solution and marks a shared flag. Each thread periodically checks this flag to determine if it needs to continue its search. If a thread reaches a contradiction or sees that a solution has been discovered by another thread, it decrements a thread counter and terminates. The last thread to complete prints the result (satisfying solution or unsatisfiable) and associated statistics.

The MTA-2 is unable to automatically extract this type of dynamic parallelism. Therefore, the future construct is used to create new threads during execution. However, because each processor can support only a limited number of active threads, using a future to create a new thread at each decision point quickly results in numerous futures waiting for execution. The overhead involved in maintaining the list of available futures as well as the memory consumed by the futures and their associated data structures becomes prohibitive for large benchmarks. In order to avoid an explosion of futures waiting for execution, a child thread is created only if the current thread counter is less than some predefined limit. Otherwise, the original thread executes sequentially, following the second branch itself. The value of this limit depends on the characteristics of the architecture.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SAT?</th>
<th>Variables</th>
<th>Clauses</th>
<th>M/N</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Uniform Random 3-sat</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uf20-01</td>
<td>SAT</td>
<td>20</td>
<td>91</td>
<td>4.55</td>
</tr>
<tr>
<td>uf50-01</td>
<td>SAT</td>
<td>50</td>
<td>218</td>
<td>4.36</td>
</tr>
<tr>
<td>uuf50-010</td>
<td>UNSAT</td>
<td>50</td>
<td>218</td>
<td>4.36</td>
</tr>
<tr>
<td>uuf50-011</td>
<td>UNSAT</td>
<td>50</td>
<td>218</td>
<td>4.36</td>
</tr>
<tr>
<td>uuf50-012</td>
<td>UNSAT</td>
<td>50</td>
<td>218</td>
<td>4.36</td>
</tr>
<tr>
<td>uuf50-013</td>
<td>UNSAT</td>
<td>50</td>
<td>218</td>
<td>4.36</td>
</tr>
<tr>
<td><strong>Planning</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blocksworld-med</td>
<td>SAT</td>
<td>116</td>
<td>953</td>
<td>8.2155</td>
</tr>
<tr>
<td><strong>All Interval Series</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ais6</td>
<td>SAT</td>
<td>61</td>
<td>581</td>
<td>9.5246</td>
</tr>
<tr>
<td>ais10</td>
<td>SAT</td>
<td>181</td>
<td>3151</td>
<td>17.409</td>
</tr>
<tr>
<td>ais12</td>
<td>SAT</td>
<td>265</td>
<td>5666</td>
<td>21.381</td>
</tr>
</tbody>
</table>
For the MTA-2 the number of threads that can be active is restricted to 128 per processor. Therefore, the thread limit is set to 100 threads per processor. Even with this limit on the number of threads/futures, memory quickly becomes an issue for large benchmarks. This problem could be partially eliminated by finding more efficient methods of managing data structures and reducing copying. Also, a more complicated run-time load balancing algorithm that takes into account the memory load as well could be more efficient.

The SALT simulation architecture is able to efficiently capture this type of dynamic parallelism using DimC’s threaded function calls. Because the thread management overhead on the lightweight multithreaded architecture is lower than that of futures, the DimC solver implementation creates a new thread for each decision point and for each clause evaluation, with no restriction on the total number of threads in the system. For the clause evaluation, data is shared among threads within a decision point. Operations on shared data, such as assigning a value to a variable, inserting on a list, or removing from a list, use extended memory semantics to avoid conflicts during parallel evaluation of the clauses.

6.5 Results

The multithreaded SAT solver was executed on the MTA-2 and in the SALT simulation environment to explore different performance issues. The MTA-2 version extracts parallelism at the decision points and restricts the number of threads to 100 per processor. In contrast, the SALT version extracts parallelism at both the decision point and clause evaluation level and does not restrict the number of threads. The solver was executed for a set of both satisfiable and unsatisfiable benchmarks. Multiple runs of each was done in order to reduce the variance in
execution time. In order to validate the solver code, a C program was written to check the correctness of a satisfying assignment determined by the solver. This program applied the satisfying assignment to the original formula to check that it did indeed satisfy the formula. For unsatisfiable benchmarks, the search level and number of decision points was used to provide a “sanity check” that the solver had likely explored the entire search space. The remainder of this section presents a discussion of the performance results on both the MTA-2 and LWP platforms.

6.5.1 MTA-2 Results

The multithreaded SAT solver was implemented on the MTA-2 with a limit of 100 threads per processor. Table 6.3 presents the parallel speedup results for a set of satisfiable benchmarks. The parallel speedup depends not only on the amount of parallelism extracted but also on the amount of work done. Typically when a benchmark is parallelized across $P$ processors, the total amount of work done remains fairly constant with each of the processors executing approximately $\frac{1}{P}$ of the workload plus some additional overhead for going parallel. However, for satisfiable problems different runs may search the tree in a different order, yielding search paths of different lengths and thus varying the total workload.

Figure 6.3 shows the performance characteristics for the satisfiable benchmarks. In order to quantify the workload, the maximum level reached in the tree (a) and the total number of decision points (b) required to solve the problem were tracked for each benchmark. The parallel speedup is presented in subfigure (c). Subfigure (d) shows the fraction of processor cycles able to issue an instruction, reflecting processor utilization. The average number of streams per processor (e) represents the amount of parallelism extracted from the benchmark. Finally,
TABLE 6.3

SPEEDUP FOR SATISFIABLE BENCHMARKS ON MTA-2

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>uf20-01</td>
<td>1.43</td>
</tr>
<tr>
<td>uf50-01</td>
<td>0.54</td>
</tr>
<tr>
<td>blocksworld-med</td>
<td>1.88</td>
</tr>
<tr>
<td>ais6</td>
<td>1.07</td>
</tr>
<tr>
<td>ais10</td>
<td>3.47</td>
</tr>
<tr>
<td>ais12</td>
<td>1.50</td>
</tr>
</tbody>
</table>

the average number of memory references per processor per cycle, subfigure (f), captures the memory usage.

In order to understand the parallel speedup, we look primarily at the number of decision points (i.e. work) and the number of streams (i.e. parallelism). In general, more decision points yields more streams. However, this is not always the case. The poor speedup shown by uf50-01 is due to a large increase in the number of decision points as the number of processors increases. Although the amount of parallelism extracted is high, at almost 100 streams per processor, it is not enough to overcome the large increase in the amount of work. Moderate speedup is shown for the uf20, blocksworld-med, and ais6 benchmarks which all show slight increases in the number of decision points and decreases in the number of streams per processor as more processors are added. Finally, for both the ais10 and ais12 benchmarks, the number of decision points decreases as the number
Figure 6.3. Performance Characteristics for Satisfiable Benchmarks on MTA-2
of processors increases. This, coupled with a high level of extracted parallelism, provides superlinear speedup overall. The multithreaded SAT solver was able to show parallel speedup for all satisfiable problems, even when the amount of work increased with the number of processors.

In order to further quantify performance, a set of unsatisfiable benchmarks was executed on the MTA-2. The speedup results for these benchmarks are shown in Table 6.4. Figure 6.4 shows the parallel speedup (a), issue rate (b), number of streams (c), and number of memory references (d). For these benchmarks, the total number of decision points and the maximum search level remain constant across runs because the entire space must be searched. Therefore the speedup is determined primarily by the amount of parallelism extracted. The unsatisfiable benchmarks show much more consistent results with speedups ranging from 7 to 11 on 16 processors. Better performance is associated with larger numbers of streams and good issue rates are achieved when enough parallelism exists to extract 80-100 threads per processor. It is also interesting to note the high level of memory usage associated with both the satisfiable and unsatisfiable benchmarks. The number of memory references per cycle are just slightly lower than the issue rate, indicating that a memory reference is issued on almost every cycle.

6.5.2 LWP Simulation Results

The multithreaded SAT solver was also ported to DimC by Sheng Li and run in the SALT simulation environment. This allowed us to evaluate performance on the lightweight multithreaded architecture for both PGAS and DGAS memory models [59]. Because the lightweight multithreaded architecture supports an “unlimited” number of threads with efficient, low overhead creation and synchronization, it
Figure 6.4. Performance Characteristics for Unsatisfiable Benchmarks on MTA-2
TABLE 6.4  
SPEEDUP FOR UNSATISFIABLE BENCHMARKS ON MTA-2

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>uuf50-010</td>
<td>1.98</td>
</tr>
<tr>
<td>uuf50-011</td>
<td>2.21</td>
</tr>
<tr>
<td>uuf50-012</td>
<td>1.95</td>
</tr>
<tr>
<td>uuf50-013</td>
<td>1.97</td>
</tr>
</tbody>
</table>

is well suited to the dynamic workload presented by this problem. In addition, threaded function calls allow threads to be efficiently created and terminated dynamically during execution. The DimC implementation extracts parallelism at both the decision point and the clause evaluation level. At each decision point, a new thread is created and data is copied, with only the formula being shared globally. For the clause evaluation, a thread is created to evaluate each clause and data is shared among the threads within a decision point. For the DGAS memory model, data and threads are distributed throughout the system in a round robin fashion, effectively eliminating any locality. In the PGAS memory model, threads and data are collocated at creation to ensure locality. This is done using DimC’s threaded function call with a memory pointer parameter that creates the thread at the location indicated by the pointer.

Figure 6.5(a) shows the SAT solver performance for the DGAS mapping. Linear speedup is achieved for up to 8 processors. After that point performance diminishes because there is no longer enough parallelism available to keep the
processors saturated. This performance is notably different from that achieved on the MTA-2, shown in Figure 6.5 (c). The MTA-2 results show significantly smaller speedups along with more variable performance. The LWP architecture shows better speedup primarily because its hardware support for large numbers of threads and dynamic thread management offer lower overhead than the MTA-2’s future construct. The variability shown by the MTA-2 is due primarily to differences in thread execution ordering. The ordering of execution in the simulation environment is more deterministic than on the MTA-2, yielding less variability in the workload for the satisfiable problems. The unsatisfiable problem, uuf50-218-1, does not suffer from this variability in workload. For this benchmark, speedup on both architectures increases up to 8 processors and then decreases after this point.

Figure 6.5 (b) shows speedup for the LWP simulations under the PGAS model. PGAS provides better performance than DGAS, with almost linear speedup for up to 16 processors. The DimC language and compiler are designed to provide efficient support for dynamic affinity between threads and data as shown by this problem. This is achieved by supplying a memory pointer as a parameter during dynamic thread creation. This memory pointer is used to assure that the thread is collocated with the data at the specified address. Without this type of support, programming applications to take advantage of this dynamic affinity would be extremely difficult.

6.6 Key System Characteristics to Support the Multithreaded SAT Problem

This section synthesizes the multithreaded SAT solver results with the results of previous experiments to determine both the programming and architectural
Figure 6.5. Comparison of SAT solver speedup on LWP and MTA-2
characteristics needed to support optimum performance for the multithreaded SAT solver.

The structure of the SAT solver is such that it provides a large degree of dynamic parallelism. This requires programming paradigms to effectively identify and extract dynamic parallelism as well as architectural mechanisms to support the dynamic creation, execution, and termination of threads. Parallelism in the SAT solver is utilized at both the decision point level and during clause evaluation. During execution, a thread is created at each decision point and for the evaluation of each clause on the clause evaluation list. This type of parallelism cannot be scheduled statically and requires an efficient programming mechanism for the dynamic creation of threads. The threaded function call found in DimC provides just such a mechanism. It creates a new thread for each function invocation, allowing the programmer to easily capture both the recursive parallelism of the decision points and the list based parallelism of the clause evaluation. Capturing this dynamic parallelism is key for performance of the multithreaded solver.

The multithreaded SAT solver also shows dynamic data/thread affinity that can be difficult to program for many traditional parallel architectures. However, this affinity between a thread and its data is efficiently extracted from the code using the thread locality parameter associated with the threaded function call. This thread locality mechanism allows the solver to achieve superior performance when the memory system utilizes a partitioned global address space (PGAS). In the multithreaded SAT solver, a new thread and copies of the main data structures are created at each decision point, with only the formula data structure shared among decision points. To achieve optimum performance, the system must be able to leverage this affinity between a thread and its data by optimizing locality. The
locality parameter associated with the threaded function call enables the runtime to collocate the data and thread, taking advantage of the PGAS memory system to minimize the memory latency. Furthermore, a runtime is needed to distribute the placement of threads and data to maintain load balance across the system.

In addition to efficiently extracting dynamic parallelism and thread/data affinity, large numbers of dynamic threads must be efficiently managed with very low overheads. The proposed architecture has very lightweight threads with minimal associated state that can be created and terminated with only a few instructions. By allowing threads to continue execution after issuing a memory reference, blocking occurs only when the result of the reference is needed but is unavailable. This allows threads to continue executing longer before blocking, thereby reducing the frequency of eviction of blocked thread frames from the frame buffer. Additionally, an eviction policy should be implemented that attempts to minimize swapping while evicting thread frames that are likely to have long latencies to allow new threads to execute. Minimizing evictions reduces the thread overhead as well as the pressure on memory. Excessive memory pressure can cause congestion in the network and at memory banks, increasing memory latencies. Rather than improving performance, increasing the number of threads can further aggravate this situation. Given the memory intensiveness of the application, it is important that the memory system support one memory request per processor per cycle for optimum performance.

Efficient management of thread synchronization and data sharing, provided by the system’s extended memory semantics, is another key component of performance. Threads must coordinate execution, notifying other threads when a contradiction or solution is found, as well as share data structures among threads.
during clause evaluation. Extended memory semantics, primarily full/empty semantics, provide an efficient mechanism for operations on shared data such as setting or checking flags, assigning values to shared variables, and inserting to or deleting from shared lists. Adding optimizations to the solver such as the sharing of learned clauses would further increase the amount of data sharing and synchronization in the application making its efficient implementation even more important. In addition, the extended memory semantics are used to maintain execution ordering within threads and provide low overhead blocking and unblocking in the face of multiple outstanding memory references.

Finally, the system does not utilize a traditional data cache because of the interference effects of large numbers of threads and the difficulty in maintaining coherency for shared memory. The cost of eliminating caches is offset by both the use of embedded memory to shorten latencies and multithreading to tolerate higher latencies.

6.7 Conclusions

The SAT solver application presents a widely applicable problem that is difficult to parallelize on traditional architectures. A multithreaded implementation of the core BCP routine is executed on both the MTA-2 and the SALT simulation environment. Results identify a number of characteristics of the proposed architecture that enable it to provide good performance. Although the multithreaded SAT solver has a large degree of parallelism, it is very dynamic and difficult to capture. In order to perform well, the programming language must have mechanisms to efficiently extract dynamic parallelism. Both the MTA-2’s future construct and DimC’s threaded function calls provide this functionality.
However, the overhead involved in managing large numbers of dynamic threads is much lower for the proposed architecture’s threaded function calls, demonstrating better performance than the MTA-2’s future implementation, as shown in Figure 6.6 for the uuf50-218-1 benchmark. Although the solver demonstrates a very irregular workload arising from the searching nature of the problem, threads created at each decision point have a high affinity to the data structures created with them. The locality parameter associated with DimC’s threaded function call combined with a PGAS memory model allows the architecture to capture and utilize this affinity between threads and data to increase the locality of memory references. Finally, efficient thread synchronization and data sharing provided by extended memory semantics are essential to parallel performance as the number of threads increases.

Both the DimC and MTA-C implementations of the multithreaded SAT solver achieved good performance with reasonable programming complexity. Additional performance gains could be realized by incorporating optimizations into the code such as learning and decision strategies, by exploring more efficient data structures, and by utilizing a runtime to more efficiently manage the creation and distribution of threads.
Figure 6.6. Comparison of Multithreaded SAT Solver Speedup on MTA-2 and LWP Architecture for uuf50-218-1
CHAPTER 7

PUTTING IT ALL TOGETHER: CONFIGURING THE ARCHITECTURE AND APPLICATIONS FOR PERFORMANCE

This chapter synthesizes the results of previous chapters to propose a baseline node configuration and discusses how to program applications to take advantage of the architecture’s unique features. Section 7.1 discusses the preferred architectural configuration based on experimental results. Section 7.2 then discusses application programming for the architecture. Section 7.4 describes the system’s expected performance advantages over other architectures, particularly the Blue Gene/L. Finally, Section 7.3 concludes by summarizing what has been learned about the architecture, how to program for it, and the advantages it provides.

7.1 Lightweight Multithreaded Architecture

The lightweight multithreaded architecture evaluated in this work consists of a network of lightweight processors and memories. The lightweight processors support large numbers of lightweight threads with minimal state represented as data frames in memory. The memories utilize extended memory semantics to support low-cost thread management and memory-based synchronization. Together these features change the way in which applications are programmed, offering improved performance for many types of applications.
Multithreaded performance depends primarily on maintaining enough threads to keep the processors busy. As system sizes increase, the number of processors, the memory latency, and the memory bandwidth requirements all increase as well. This yields a corresponding increase in the number of threads needed to mask the memory latency and saturate the processors. In order to support large numbers of threads, the threads themselves must be efficiently created, managed, and synchronized. This need drives many of the architectural tradeoffs under exploration here. A number of specific issues were explored for the baseline architecture, as described in Chapter 3, to determine their effects on performance and each is discussed in turn.

What are the system bandwidth requirements and what configuration is needed to support them? Specifically, how many processors and eDRAM banks should there be per LPC? Although multithreading is very effective at tolerating memory latency, it cannot solve all memory ills. Experiments demonstrated that memory bandwidth is crucial to performance, especially for data-intensive applications. If the memory system cannot provide sufficient bandwidth, congestion in the memory increases the overall memory latency, negatively impacting performance. In the case of a bandwidth-limited system, increasing either the number of threads per processor or the number of processors increases the number of memory accesses and thus the bandwidth requirements, worsening performance rather than improving it as might be expected. This indicates that multithreading alone cannot overcome an inferior memory system. Therefore, the system must be configured to support the bandwidth requirements of the target applications.

Experiments examined the bandwidth requirements for a variety of system configurations and application characteristics to determine a baseline configura-
tion, shown in Figure 7.1. Ideally for memory intensive applications, the memory system should be able to handle at least one memory reference per processor per cycle. Results indicated that 16-64 eDRAMs are needed per processor to avoid congestion in the memory system and achieve high processor utilization. These values hold for 1-4 processors per LPC and 4 DRAMs per LPC. For greater than 4 processors per LPC, congestion at the DRAMs becomes a problem. The system’s interconnect must also be able to handle the traffic generated. For a system configuration of 4 processors, 64 eDRAMs, and 4 DRAMs, results show that the on-chip interconnect must handle approximately 1650 bytes/cycle and the memory interconnect must handle approximately 18 bytes/cycle. The LPC to LPC network traffic depends on the number of LPCs in the system and the distribution of accesses. Assuming the system contains N LPCs and no locality, \((N - 1)/N\) of the accesses will be remote, yielding an estimated bandwidth requirement of \(1650 \times (N - 1)/N\) bytes per LPC per cycle for the system network. Further research using cycle-accurate simulations are needed to explore this issue. This analysis, however, provides a good starting point for these investigations.

Should a thread block immediately on a memory reference or should non-blocking memory references be supported? Results showed that supporting non-blocking memory references improves performance by allowing threads to execute longer before blocking, thereby reducing the number of threads needed to mask a given latency and saturate the processor. Performance continued to improve as the number of concurrent memory references allowed was increased. For more than 2-4 outstanding memory references, however, the improvements were relatively small. Because each thread executing on a given processor may make a memory reference, the system is already required to support multiple outstanding memory
Figure 7.1. Baseline LPC Configuration
references for each processor and to have a mechanism in place to notify a thread when its reference has completed. Multiple, non-blocking memory references for each thread can be supported by extending these same basic mechanisms. It is important to note that non-blocking memory references may also result in an increase in the number of references per processor and thus the memory bandwidth requirements, reducing performance in a bandwidth-limited system as discussed previously.

When should a blocked thread be evicted to memory from the frame buffer? In addition to memory references made by the application, the management of thread frames in memory also consumes memory bandwidth. Experiments showed that thread frame evictions add to the traffic at memory. Allowing thread frames to selectively block in the frame buffer, rather than evicting them to memory immediately upon blocking, reduces evictions and improves performance. Furthermore, performance increases as the number of threads allowed to block in the frame buffer is increased. The eviction scheme in this work defined a frame buffer block limit and evicted any thread that caused this limit to be exceeded. This simple eviction policy provides a reasonable design point from which to investigate the system wide effects of the frequency of thread frame eviction but more complex heuristics would likely result in better performance. Ideally, threads that block for long periods of time would be evicted to memory while those that block for short periods of time would remain in the frame buffer to avoid thrashing. Further research is needed to determine heuristics to intelligently choose which threads to evict to memory while guaranteeing fairness, avoiding deadlock, and keeping the overhead low enough to support large numbers of threads.

Should a PGAS or DGAS memory addressing model be used? Although multi-
threading is very effective for tolerating memory latencies, longer latencies increase the number of threads needed to saturate the processors and increase execution time if the number of available threads is insufficient to fully mask those latencies. One approach to reducing such latencies is to configure the architecture to use a PGAS memory addressing model to take advantage of data locality rather than a DGAS model that distributes data to eliminate locality. Results showed that for applications that exhibit locality, the PGAS model requires $1/2$ to $1/3$ the number of threads to saturate a processor as the DGAS model. Therefore for the same number of threads, the PGAS model provides significantly better performance. Programming an application to take advantage of data locality, however, can be extremely complex and time consuming and depends greatly on the programmer’s ability to decompose the work and data effectively. Furthermore, many applications do not exhibit significant locality or the locality is extremely difficult to extract. In these cases, changing the focus to extracting more parallelism rather than increasing locality may achieve similar or even improved performance. To support optimum performance, the memory should support a PGAS model to control data placement and take advantage of locality where it is available. The choice of memory addressing model influences various components of the architecture as well as the compiler, runtime, and features of the programming language. Further research into these areas is needed.

Multithreaded performance depends on having enough threads available to keep the processor utilized and mask the memory latency and this section described an architectural configuration to support this. The next section will discusses how to program applications to take advantage of this architectural configuration.
7.2 Multithreaded Application Programming and Execution

In evaluating an architecture, it is important to consider not just peak performance but the performance of specific applications and the programming complexity required to achieve it. Recently, the term productivity has been coined to address this, referring to both programming effort and performance in evaluating a system. The previous section described how the architecture should be configured to support performance. This section addresses other aspects of productivity, discussing how applications can be programmed to take advantage of the architecture’s features and the programming mechanisms and tools needed to do so efficiently.

How should applications be structured to best utilize the architecture? Conventional architectures are typically designed to support coarse grain parallelism composed of a small number of mostly independent processes/threads. Traditional thinking purports that a program should be divided into large blocks of execution and that data sharing and other synchronization are to be avoided. This is because, for conventional architectures, process/thread creation and management as well as data sharing and synchronization are very expensive. In contrast, the lightweight multithreaded architecture is specifically designed to efficiently support fine-grain parallelism consisting of many small threads of execution with frequent data sharing and thread synchronization. This changes the way in which the architecture is programmed, capturing previously unavailable parallelism.

Many applications are well suited to the coarse-grain programming structure used on conventional architectures. Other applications, however, exhibit parallelism that is difficult to extract or other characteristics that limit their performance on conventional architectures and, in these cases, the proposed architecture
can often deliver superior performance. Experiments explored a variety of applications that exhibit a range of characteristics including frequent data sharing, hot spots, barrier synchronization, and dynamic workloads, all of which may limit performance on conventional architectures. Different application characteristics place different demands on the system and require a range of programming approaches, each of which will be described in turn.

In some applications, parallelism is available but not usable because it is too fine-grain for the architecture. If the amount of work in a thread isn’t sufficient to balance the overhead involved in creating and managing the thread, the parallelism is not realizable. This may occur when there are too few iterations of a loop or the amount of work within an iteration is too small to outweigh the cost of going parallel. The lightweight multithreaded architecture reduces this problem by offering very low overhead thread creation and management to increase the amount of parallelism that can be extracted from an application.

Another typical impediment to performance is the need for synchronization, which can be very expensive. Synchronization takes many forms depending on the application’s needs. The image histogramming benchmark, for example, implemented a lock to guarantee atomic updates of the shared histogram array. The heat conduction benchmark implemented a barrier to coordinate thread execution between iterations of the heat computation. The FFT butterfly graph benchmark used full/empty bit semantics to implement producer/consumer like exchanges of data. EMS support for efficient thread blocking and memory-based synchronization makes these operations very efficient, even for large numbers of threads, allowing them to be used much more frequently.

An application may also exhibit a lack of locality or have locality that is diffi-
cult to capture. Because of the cache-based nature of conventional architectures, significant effort is put into maintaining locality while extracting parallelism and those applications without locality perform very poorly. The heat conduction benchmark, for example, required complex data and thread distributions in order to utilize locality. Other applications may have little or no locality such as those with random memory accesses or “pointer chasing”. In the lightweight multi-threaded architecture, locality can be traded for increased parallelism to provide more threads and thus mask longer latencies. This approach of using large numbers of threads to mask latency may not only provide improved performance but may also reduce the programming complexity as well.

Finally, applications may exhibit dynamic execution, making parallelism difficult to capture. For some applications, such as a tree search or recursive execution as found in the NQueens and SAT benchmarks, it is impossible to know statically how many processes/threads are needed and how long they will execute. On conventional architectures this is often programmed by statically defining a set of processes/threads and assigning work to them as it becomes available or re-assigning work when the workload is severely unbalanced. The cost of maintaining work queues in software, assigning work, and load balancing can severely limit performance. When programming for the lightweight multithreaded architecture, threads can be efficiently created when work is available and destroyed when no longer needed. The architecture supports large numbers of concurrent threads and these threads can be distributed at creation to balance the workload. Furthermore, the architecture provides support for dynamic affinity between threads and data that allows a thread to be assigned to a processor that is close to the data on which it will operate, leveraging locality that can be difficult to capture
Other dynamic applications, such as the FFT butterfly graph benchmark, consist of a static number of total threads with complex interactions that make it difficult to determine a static schedule. These types of applications benefit from a data-driven approach where a set of threads is defined that interact through the exchange of data. This approach utilizes synchronization primitives to control thread scheduling and can reduce programming complexity, increase the amount of parallelism extracted, and improve performance.

What programming paradigms and tools are needed to effectively support multithreaded application development and performance for the lightweight multithreaded architecture? Specifically, how should thread creation and management, data and workload distribution, and synchronization be handled? In order to achieve good performance on the lightweight multithreaded architecture, many applications demand dynamic parallelism with large numbers of threads, frequent data sharing, and synchronization. These application demands require programming paradigms and tools to support these operations. This work explored application development and execution for different programming languages and platforms to examine a variety of parallel programming paradigms and understand the compiler and runtime issues involved in developing multithreaded applications for the lightweight multithreaded architecture.

At the lowest level, the compiler should attempt to automatically extract parallelism inherent in the application without the need for the programmer to explicitly identify and partition it. The MTA-2 compiler demonstrated the ability to analyze a program to determine the number of threads needed as well as to automatically identify parallelism and distribute it among threads. This is par-
icularly effective for loop-level parallelism. Many applications, however, exhibit parallelism that cannot be easily identified by a compiler. This may be because it is difficult to resolve dependencies in the loop or because the parallelism takes other forms such as recursion or independent functions. In these cases the compiler needs assistance from the programmer to identify and distribute parallelism. This can take many forms, ranging from pragmas or directives to parallel control structures to explicit programmer defined threads. Each of these has its place.

Pragmas or directives are not explicitly part of a programming language but act as programmer hints to the compiler to help identify parallelism or control the distribution of such parallelism including data and threads. On the MTA-2 different types of directives assist the MTA-C compiler in different ways. For example, directives may help to identify parallelism by indicating that a loop should be parallelized or has no dependencies or may help to better distribute that parallelism by suggesting the number of threads to use or how to distribute the iterations among the threads. In contrast, parallel control structures are explicit parallel language constructs that are similarly used to identify parallelism and its distribution. UPC, for example, uses parallel constructs such as forall to indicate that a loop is parallel and indicate the distribution of iterations among the threads. Furthermore, it allows the user to define whether a data structure is private to the thread or shared and, for shared data, to suggest the best way to distribute the data throughout the system. DimC does not have a parallel loop construct but it allows the user to indicate whether a data structure should be placed in the thread’s frame or in global memory, and data in all memory locations can be shared, including those in thread frames. Support for some combination of pragmas/directives and parallel control structures is needed to help identify and
distribute parallelism on the lightweight multithreaded architecture.

For many applications, particularly those with dynamic, recursive type parallelism, more explicit control is needed such as mechanisms for programmer-directed thread creation and distribution. The degree to which this is allowed varies greatly among platforms and languages. UPC, for example, requires the user to define the parallelism statically at runtime. It uses a single program multiple data (SPMD) model where each thread executes the same program, although each thread may take a different path through the program during execution. The threads are defined at run time and the number of threads cannot change during execution. This approach makes programming complex and, for applications like the NQueens, it can require extra work during execution.

The MTA-2 provides significantly more dynamic threads. The compiler determines how many hardware streams are needed, and they are reserved at runtime. The number of streams cannot change during execution. The number of threads, however, can change during execution, and they are assigned to available streams for execution. The MTA-C programming language provides the future programming construct to specify that a sequence of code can be executed by an independent thread. The programmer is not able to control how the threads are distributed. The compiler and runtime are responsible for this. Although the future construct supports the dynamic creation of threads, it manages them in software, making it too expensive to support the large numbers of dynamic threads needed for application performance in the lightweight multithreaded architecture.

The DimC language, developed for the LWP simulation platform, addresses these issues through the use of a threaded procedure call. Through the threaded procedure call, the programmer explicitly defines a thread for each invocation of
the procedure. In this way, large numbers of threads can be created dynamically throughout execution. Much of the thread creation and management is done through hardware-managed lists. However, this still puts significant pressure on the runtime to distribute the threads to maintain load balance. Furthermore, creating more threads than are needed to saturate the processors can sometimes adversely affect performance. Therefore, it would be useful for the programmer to be able to indicate that new threads should be created as needed to saturate the processors and, after that point, existing threads can handle any additional work. The threaded procedure call also has support for affinity between data and thread by providing a locality parameter which indicates the “neighborhood” of execution. This indicates to the runtime that the thread should execute on a node whose associated memory contains the data indicated by first parameter. Therefore, further research is needed into the development of efficient runtime support for thread creation and the distribution of threads and data.

Finally, languages need specific support for synchronization. The language should provide some common higher-level synchronization mechanisms, such as barriers and locks as found in UPC. These mechanisms can be efficiently built using the system’s extended memory semantics. It is also important, however, that the language provide direct access to the extended memory semantics. These can be used by applications in a number of ways, such as using full/empty bits to efficiently ensure memory ordering consistency in producer/consumer type exchanges. Further research is needed into how to build higher level mechanisms using the EMS as well as how to most efficiently support EMS in hardware.
7.3 Performance Advantages

The lightweight multithreaded architecture explored in this work has a number of unique architectural features that differentiate it from other systems and change the way in which applications are programmed. The architecture is composed of a network of lightweight processors and memories that provide hardware support for large numbers of lightweight threads and extended memory semantics to implement efficient memory-based thread management, data sharing, and synchronization. These features make the architecture well suited to applications with large numbers of dynamic threads, data sharing, and intense synchronization, characteristics which challenge many conventional architectures. As mentioned previously in Chapter 2, the IBM Blue Gene/L held the number one spot in the Top 500 Supercomputers list until recently, making it an important architecture for comparison. This section examines the types of applications for which the lightweight multithreaded architecture would be expected to perform better than the massively parallel Blue Gene/L.

The Blue Gene/L (BG/L) is a massively parallel architecture designed to provide an exceptional cost/performance ratio for a set of applications with good scaling behavior [41]. It achieves this performance by using large numbers of processors of moderate frequency, integrated using system-on-a-chip technology. The BG/L has a distributed memory and uses a message passing model with MPI for programming. Although a message passing model can be useful for applications with coarse-grain parallelism, it can be more difficult to program than a shared memory model as used by the lightweight multithreaded architecture. Furthermore, for applications with very fine-grain parallelism, the lightweight multithreaded architecture would be expected to perform better due to the efficient...
thread management and memory-based support for synchronization. Like conventional architectures, the BG/L relies on a memory hierarchy with caches to reduce memory latencies by taking advantage of temporal and physical locality in an application. For applications that do not demonstrate locality, the overhead involved in caching may actually increase memory latencies. In contrast, the lightweight multithreaded architecture can achieve better performance by leveraging large numbers of threads to mask the memory latencies. This approach also simplifies programming by eliminating the need for complicated data and thread distributions to capture locality. Because of the coarse-grain nature of programming with MPI and its associated overheads, the lightweight multithreaded architecture would be expected to perform better for those applications that demonstrate very dynamic behavior that cannot be statically partitioned. Furthermore, although the BG/L provides support for small messages and multiple communication networks to provide efficient performance for common operations like collectives and barriers, MPI and the architecture are ill-suited to very irregular and dynamic communication and synchronization that cannot be scheduled statically. Therefore, for certain classes of applications, the lightweight multithreaded architecture would be expected to perform better than the BG/L.

7.4 Conclusions

Multithreaded performance depends on having enough threads available to keep the processor utilized and mask the memory latency. Thus, the performance of applications on the lightweight multithreaded architecture can be improved either by decreasing latency or increasing parallelism and this need drives the architectural tradeoffs discussed here. Increasing the number of processors, increasing
the number of threads per processor, or allowing threads to issue non-blocking memory references effectively increases the amount of parallel execution. However, each of these may also increase the amount of memory traffic, resulting in an increase in memory latencies and negatively impacting performance if the system cannot support the required bandwidth. This issue is addressed by analyzing the system’s bandwidth requirements and determining the configuration of processors, eDRAMs, DRAMs, and networks needed to support them. Furthermore, allowing threads to block inside the frame buffer reduces the number of evictions to memory, reducing the memory traffic. Memory latencies can also be reduced by supporting a PGAS memory addressing model to take advantage of data locality. Combined, these approaches define the architectural configuration needed to support application performance.

The lightweight multithreaded architecture is specifically designed to support fine-grain parallelism consisting of many small threads of execution with frequent synchronization and data sharing. The unique features of this architecture change the way in which applications are programmed, capturing previously unavailable parallelism. Low overhead thread creation and management support very fine-grain parallelism to increase the amount of parallelism that can be extracted from an application. EMS support for efficient thread blocking and memory-based synchronization make operations such as barriers, locks, and producer/consumer data sharing less expensive, allowing them to be used frequently by applications. Multithreading allows applications to be implemented using large numbers of threads to mask memory latencies rather than relying only on complex data and thread distributions to attempt to reduce memory latency through locality. Finally, the lightweight multithreaded architecture is well suited to applications with dynamic
execution. Large numbers of dynamic threads can be created and destroyed as needed during execution and threads can be collocated with the data on which they operate to take advantage of the affinity between threads and data. Furthermore, applications with dynamic execution that cannot be scheduled statically benefit from a data-driven approach where synchronization primitives are used to control thread scheduling. This approach can reduce programming complexity, increase the amount of available parallelism, and improve performance.

In order to support application development, programming paradigms and tools are needed to support dynamic parallelism with large numbers of small threads, frequent synchronization, and the sharing of data among threads. Compilers for the architecture should be able to extract simple parallelism, such as loop level parallelism, automatically. Programming languages should support pragmas/directives or parallel control structures to help identify more complex parallelism and direct data and thread distribution. Programming language and runtime support for explicit thread creation and distribution, such as that found in the DimC threaded procedure call, is also important. Finally, languages must provide both high level synchronization mechanisms such as barriers and locks and access to the low level extended memory semantics to support full/empty semantics and other types of synchronization.

Together, the programming paradigms and tools allow applications to take advantage of the architecture’s unique features, capturing previously unavailable parallelism and improving performance for certain classes of applications. In particular, the lightweight multithreaded architecture would be expected to perform better than the Blue Gene/L for applications with very fine-grain parallelism, limited locality, dynamic execution, or intense synchronization.
CHAPTER 8

CONCLUSIONS

As conventional parallel architectures reach the limits of their scalability, revolutionary new architectures are needed in order to achieve the performance level demanded by increasingly large and complex applications. This dissertation evaluates one such architecture, a lightweight multithreaded architecture, and explores how multithreaded applications can be organized to take advantage of the architecture’s unique features. It uses an organized approach to evaluation that targets the modeling, simulation, and evaluation tools to the issue under investigation, avoiding the need to simulate and build the entire machine. It then integrates these results to propose an architectural configuration and application programming model. This comprises the first attempt to code and compare real applications using the architecture’s unique architectural features.

This dissertation addresses two main questions:

1. How should the architecture be configured to be effective with applications?

2. How should the applications be organized to effectively use the architecture?

In order to answer these questions, experiments and analysis focused on understanding how key factors at the architectural and application level interact to affect performance. In Chapter 4, experiments employed analytical and simulation models with statistical workloads to examine thread management issues and
the effect of latency and bandwidth on performance for a range of architectural and application characteristics. Experiments in Chapter 5 evaluated application programming and memory models for a set of application benchmarks on both the MTA-2 and in the LWP simulation environment. Chapter 6 then evaluated the programming and performance of a multithreaded satisfiability solver for the lightweight multithreaded architecture. The key insights developed from these explorations were then combined in Chapter 7 to define a baseline architectural configuration, describe how to program applications to utilize the architecture’s unique features, and describe the expected performance advantages.

This work contributes to the understanding of how architecture and application characteristics affect performance on the lightweight multithreaded architecture and what this means for the architectural configuration and the organization of applications. The key contributions of this research include the definition of a baseline node configuration and an approach to application programming that utilizes the unique features of the architecture to capture previously unavailable parallelism. The baseline node configuration specifically addresses the bandwidth requirements of the system and the node organization needed to support them as well as the performance advantages provided by supporting multiple non-blocking memory references per thread, a thread eviction policy that allows threads to selectively block in the frame buffer, and a PGAS memory model. The application programming model achieves performance on the lightweight multithreaded architecture by leveraging fine-grain and dynamic parallelism, shared data, frequent synchronization, and the affinity between threads and data. Furthermore, it proposes the programming paradigms and tools needed to efficiently support application programming. The remainder of this chapter summarizes the key
insights of this research and opportunities for further research.

8.1 Lightweight Multithreaded Architecture

In Section 4.5, analytical and simulation experiments explored the effects of allowing multiple, non-blocking memory references per thread. Results showed that non-blocking memory references increase the amount of memory access latency that can be hidden before the performance becomes limited by communication. This effectively reduces the number of threads needed to saturate a processor and improves performance in the communication limited region. Specifically, the analytical model results indicated that, for a system with unlimited bandwidth, allowing two outstanding references doubles the latency that can be masked. Simulation model results, however, found that for a system with limited bandwidth, congestion in the memory significantly increases overall latencies, making the system communication-limited almost immediately.

Although results showed that multithreading is very effective in overcoming the latencies involved in traversing the network and accessing memory, increased multithreading does not always improve performance. If the processors are all saturated, increasing the number of threads does not improve performance and may even adversely impact performance, due to the overhead of thread creation and management. Performance may be improved, however, by increasing the number of processors if the number of threads is greater than that needed to saturate the current processors. Simulation results also demonstrated that increased multithreading is not as effective when memory congestion is the cause of the extended latencies. Specifically, for the analytical model with unlimited bandwidth, increasing from $2^{10}$ to $2^{12}$ threads reduced execution time by approximately 70%
while for the simulation model with insufficient bandwidth, reductions in execution time were only about 30%. Examination of memory queues provided insight into this effect showing that, although increasing the number of threads or the number of non-blocking memory references per thread increases the latency that can be masked, it also increases the number of memory references. This yields an increase in the congestion at memory, increased overall latency, and reduced performance. These results underscored the importance of understanding the system’s bandwidth requirements and configuring the architecture to support them. Furthermore, for a bandwidth-limited system, results indicate that performance is achieved by balancing the need to keep the processor busy with the need to limit congestion in the memory system.

Section 4.6 examined the thread frame eviction policy to understand the effect of thread frame evictions on performance and to determine whether threads should be allowed to block with their frames remaining in the frame buffer. Because the analytical model was unable to capture the effects of congestion in the memory, only simulation was used for evaluation of the thread frame eviction policy. Results showed that swapping threads between the frame buffer and memory adds to the congestion at memory, causing a significant drop in performance. For example, increasing the number of threads from 128, where they all fit in the frame buffer, to 256 resulted in a drop in processor issue rate of up to 60% if the processor is not saturated. Allowing threads to selectively block in the frame buffer rather than be immediately evicted reduces the amount of memory traffic generated and improves the issue rate. For a configuration with 4 processors and 4 eDRAMs the issue rate more than doubles when allowing up to 75% of the thread frame buffer’s threads to block before eviction. These results further highlighted the importance
of system bandwidth.

In order to address this, the number of processors and eDRAMs per LPC were varied to determine the node configuration needed to support the bandwidth requirements. Results indicated that for 1-4 processors per LPC, 16-64 eDRAMs per processor provide an issue rate of approximately 1 (i.e. 100% processor utilization) when all threads fit into the frame buffer and an issue rate ranging from 0.6 to 1 for larger numbers of threads where swapping between the buffer and memory is required.

The issue of the memory model was then explored in Section 5.3. Execution-driven evaluation, consisting of execution on the MTA-2 and execution in the LWP simulation environment, was used to explore the effects of different memory models on performance. Experiments compared a PGAS memory model where data and threads can be distributed to take advantage of locality to a DGAS model where data and threads are distributed without regard to locality to attempt to reduce contention and balance the workload. Results demonstrated that, for applications that exhibit locality, a PGAS model can reduce average memory latencies and significantly improve performance, requiring 1/2 to 1/3 the number of threads to saturate the processor as a DGAS model. However, programming an application to take advantage of locality can be complex and time consuming and performance depends significantly on the ability to decompose the application. For applications that do not exhibit locality or the locality is difficult to extract, a DGAS model can be simpler to program by allowing the programmer to focus on extracting more parallelism rather than increasing locality. In general, a PGAS model provides optimum performance by leveraging locality. However, in some cases, better performance may be achieved by focusing on extracting more threads
rather than attempting to capture locality.

Both Chapters 5 and 6 demonstrated the effectiveness of the extended memory semantics in supporting lightweight thread management, synchronization, and sharing of data. By employing the EMS, the architecture can support fine-grain parallelism without the performance hit incurred by conventional architectures. The histogram application showed performance improvement as the number of threads increased even though this increased the number of concurrent accesses to the shared histogram array. Similarly, performance of the heat conduction application improved with increasing number of threads even though this increased the total number of threads involved in the barrier synchronization and global reduction. Together, these results demonstrated the effectiveness of EMS for shared data and thread synchronization. The NQueens, FFT, and SAT solver also demonstrated the performance advantage provided by the lightweight multithreaded architecture’s hardware support for large numbers of dynamic threads. Specifically, the lightweight multithreaded architecture provided better speedup and could support larger numbers of threads than the MTA-2’s software-based future construct.

Section 7.1 synthesized these results to provide key insights into performance and offer a baseline node configuration based on these results. This configuration defined the node organization needed to support the system’s bandwidth requirements. It also supported multiple non-blocking memory references per thread, the blocking of threads in the frame buffer to reduce the frequency of frame eviction to memory, and a PGAS memory system to allow the programmer to take advantage of locality.
8.2 Application Programming and Performance

In Chapter 4, experiments explored architectural tradeoffs for a range of application characteristics. Results showed that for systems with sufficient bandwidth, applications can improve performance by increasing the number of threads, up to the point where all the processors are saturated. When bandwidth is limited, the application must balance the need to saturate the processors with the increased bandwidth requirements generated by additional threads. Results showed that this is especially important for memory intensive threads, which cause congestion in memory, increasing overall memory latency and reducing performance.

Programming applications to employ large numbers of relatively small threads is significantly different from the conventional parallel programming approach of distributing work into a few, coarse-grain, mostly independent processes. Chapter 5 went on to explore programming and performance for a set of application benchmarks, particularly looking at thread specification and management, work and data distribution, and mechanisms for data sharing and thread synchronization. Each benchmark illustrated different aspects of this.

In Section 5.3, the matrix vector multiply and histogram benchmarks demonstrated performance for applications with relatively simple work and data distributions. Results showed that these applications achieved 2-4 times better speedup for 64 processors when work and data were distributed to take advantage of locality, indicating that a simple parallel array programming construct would be a useful language addition. Larger problem sizes also yielded improved performance by generating more concurrent threads. Furthermore, the histogram application demonstrated the system’s ability to handle frequent updates of shared data structures. The EMS provided efficient mechanisms for atomic updates of the shared
histogram array and, even though more threads increased contention at the histogram array, the latency tolerating effects of more threads outweighed this.

The heat conduction benchmark represented an application with more complex work and data distribution. Like many other physics-based applications, it required the data to be divided into three-dimensional volumes in order to maximize locality. Even with programming language support for parallel data structures, this type of distribution can be extremely complex and time consuming to program. Conversely, programming the problem without regard for locality was much simpler but resulted in poorer performance. Results showed that, for a system with 64 processors, distributing work and data to take advantage of locality resulted in up to 2-10 times better performance than a simpler model with no locality. Furthermore, the application required a barrier synchronization and global reduction between iterations. The results showed that, unlike many conventional systems, the lightweight multithreaded architecture efficiently supports these types of global synchronization, achieving almost ideal speedup on 32 processors even with hundreds of threads per processor.

The NQueens benchmark was representative of a broad class of problems that require a search of tree-based data structures. These applications generate very dynamic execution and often show little locality, making them difficult to parallelize on conventional architectures. This type of parallelism is not automatically recognized by the compiler and requires the programmer’s explicit definition. Results showed that the threaded procedure call and hardware support for lightweight threads can efficiently support applications with large numbers of dynamic threads and offer lower overhead than a software based approach such as the MTA-2 future. Furthermore, the locality parameter provided by the threaded
procedure call was able to capture the affinity between a thread and its data within a recursive call, offering 1.5-2 times the speedup.

Taken together, these benchmark results showed how multithreaded applications can achieve performance on the lightweight multithreaded architecture by employing large numbers of dynamic threads, distributing work and data to take advantage of locality, and using shared data structures and thread synchronization, without the performance hit they often incur in a conventional architecture. Furthermore, although the MTA-2 and LWP simulation results cannot be directly compared because of differences in the number of threads and how they are managed, the MTA-2 showed better speedup than the LWP simulator for the same problem size for all but the NQueens benchmark. This is due, in part, to the MTA-2’s smaller memory access latencies but also indicates that the MTA-2’s compiler and runtime can effectively extract the threads needed to achieve good performance.

Experiments in Section 5.4 explored the implementation and scaling of active graph applications using FFT butterfly networks. Active graphs represent a class of applications typically organized as graph-like structures of nodes where execution is defined by the exchange of data between nodes. When programming for the lightweight multithreaded architecture, active graph applications can be structured to use a separate lightweight thread for each node, with the extended memory semantics used to control execution through producer/consumer like semantics. Results showed that performance improved with the number of nodes in the network, until the number of threads in the system exceeds that needed to saturate the processor. As the connectivity of the graph increases, the number of communications between threads and thus the number of memory accesses
and synchronizations increases. This requires an accompanying increase in either arithmetic operations or the number of concurrent threads to offset the increased latency. These characteristics affect the choice of implementation. Some applications can be statically scheduled in the code while still providing enough parallelism. However, for many graphs it is either impossible or time prohibitive to determine a scheduling of loops in the code that provides sufficient parallelism. Results showed that, in these cases, dynamic parallelism with the execution of large numbers of threads controlled by producer/consumer type exchanges of data is an effective way to extract parallelism.

Chapter 6 looked at the development and performance of a multithreaded satisfiability (SAT) solver and identified key system characteristics needed to support its performance. SAT solvers are applicable to a wide range of problems and have characteristics that make them difficult to parallelize on traditional architectures including a dynamic workload requiring thread synchronization and data sharing. The core component of the satisfiability solver, \( \text{BCP}() \), provided opportunities for parallelism at multiple levels. Coarse-grain parallelism can be found at each decision point by using two threads to concurrently explore the two branches of the search tree and more fine-grain parallelism can be found by evaluating different variables on the variable evaluation list concurrently or by evaluating the clauses on a given variable’s watchlist concurrently. Due to the overhead of the MTA-2’s future construct, it was only able to capture the coarse grain parallelism of the decision points and required a limit on the total number of concurrent threads in the system. The LWP simulator, however, was able to extract parallelism at the decision point and variable evaluation list levels with no restrictions on the number of concurrent threads and generally achieved better speedup when compared to
the MTA-2 results. Analysis indicated that the lightweight multithreaded architecture’s threaded support for large numbers of dynamic threads, efficient thread synchronization and data sharing, and ability to take advantage of the affinity between threads made it well suited to the multithreaded SAT solver.

All of these results were then combined in Section 7.2 to summarize how applications should be structured to best utilize the features of the lightweight multithreaded architecture and to discuss the programming tools and paradigms needed to support them. Applications achieve performance on the lightweight multithreaded architecture by

- taking advantage of fine-grain parallelism with large numbers of small threads
- using extended memory semantics to support thread synchronization and shared data
- distributing work and data to take advantage of locality when available
- focusing on increased parallelism when an application displays no locality or locality that is too difficult to extract
- using dynamic threads to capture parallelism that cannot be defined statically
- using a data-driven approach where threads interact through the exchange of data.

In order to support this, parallel programming languages need support for a combination of pragmas/directives and parallel control structures to identify and distribute parallelism, explicit definition of threads such as that provided by the threaded procedure call, and support for synchronization such as barriers, locks,
and access to the low-level EMS. The compiler should be able to automatically identify and extract simple parallelism and capture more complex parallelism with the help of the programmer. Finally, the runtime must efficiently support the dynamic creation and management of large numbers of threads.

8.3 Future Work

This work used a combination of analytical models, simulation, and execution to understand how key architectural and application characteristics interact to affect performance. Together, these can be used to direct the more detailed, cycle-accurate simulations of individual system components that are needed to build the system. Further research is needed to determine efficient thread scheduling and eviction policies as well as the size and structure of the frame buffers. More detailed evaluations of the memory system and interconnect are needed as well. Processing-in-memory (PIM) technologies should be explored for their ability to reduce latencies and efficiently implement the extended memory semantics. Interconnect technologies need to be evaluated to determine the best way to support the system's bandwidth requirements. Different caching options must be explored to determine whether the system will have data caches and what form they will take, from traditional caches to hot spot caches similar to those found in the MTA-2. These and other architectural level details provide numerous directions for further research.

The development of programming languages, compilers, and runtime software provides another fertile area for research. Programming languages must be developed that provide constructs for the explicit creation and management of threads, work and data distribution, data sharing, and thread synchronization. Compil-
ers must be developed that are able to extract parallelism, either automatically or with the help of the programmer. A runtime must be developed that can efficiently manage large numbers of explicit threads.

Finally, expanded multithreaded application development provides another region of interest. Further research is needed into which applications are best suited to the lightweight multithreaded architecture and how to program them. In order for any new architecture to gain acceptance, there must be applications for it that perform well. In particular, the current multithreaded SAT solver could be extended by incorporating optimizations into the code such as learning and decision strategies and by exploring more efficient data structures.
BIBLIOGRAPHY


