PROGRAMMING FUTURE ARCHITECTURES
DUSTY DECKS, MEMORY WALLS, AND THE SPEED OF LIGHT

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

Arun F. Rodrigues, B.S., M.S.

______________________________
Peter Kogge, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
April 2006
PROGRAMMING FUTURE ARCHITECTURES
DUSTY DECKS, MEMORY WALLS, AND THE SPEED OF LIGHT

Abstract
by
Arun F. Rodrigues

Due to advances in CMOS fabrication technology, high performance computing capabilities have continually grown. More capable hardware has allowed a range of complex scientific applications to be developed. However, these applications present a bottleneck to future performance. Entrenched 'legacy' codes – “Dusty Decks” – demand that new hardware must remain compatible with existing software. Additionally, conventional architectures faces increasing challenges. Many of these challenges revolve around the growing disparity between processor and memory speed — the “Memory Wall” — and difficulties scaling to large numbers of parallel processors.

To a large extent, these limitations are inherent to the traditional computer architecture. As data is consumed more quickly, moving that data to the point of computation becomes more difficult. Barring any upward revision in the speed of light, this will continue to be a fundamental limitation on the speed of computation.

This work focuses on these solving these problems in the context of Light Weight Processing (LWP). LWP is an innovative technique which combines Processing-In-Memory, short vector computation, multithreading, and extended mem-
ory semantics. It applies these techniques to try and answer the questions “What will a next-generation supercomputer look like?” and “How will we program it?”

To that end, this work presents four contributions:

• An implementation of MPI which uses features of LWP to substantially improve message processing throughput

• A technique leveraging extended memory semantics to improve message passing by overlapping computation and communication

• An OpenMP library modified to allow efficient partitioning of threads between a conventional CPU and LWPs — greatly improving cost / performance.

• An algorithm to extract very small “threadlets” which can overcome the inherent disadvantages of a simple processor pipeline.
# CONTENTS

**FIGURES** ............................................................. vii  
**TABLES** ............................................................... ix  
**ACKNOWLEDGMENTS** ................................................ xi  

## CHAPTER 1: INTRODUCTION ............................................ 1  
1.1 The Overall Problem ............................................. 3  
1.1.1 Hardware: Diminishing Returns ............................ 3  
1.1.2 Software: Productivity, or Programming is Hard ......... 4  
1.2 Community Goals ................................................. 6  
1.2.1 What Should the Next-Generation Supercomputer Look Like? 6  
1.2.2 What Should the Next-Next-Generation Supercomputer Look Like? ............................................. 7  
1.2.3 How Should We Program It? .................................. 7  
1.3 Dissertation Goals ............................................... 8  
1.3.1 MPI: The Now and Future King ............................ 9  
1.3.2 Relentless Parallelism: Finding Threads ................ 10  
1.4 Approach: Parallelism in New Places .......................... 10  
1.5 Scope ............................................................... 11  
1.6 Thesis Contributions ............................................ 13  
1.7 Roadmap ............................................................ 14  

## CHAPTER 2: STATE OF THE ART ...................................... 15  
2.1 Conventional Hardware ......................................... 15  
2.1.1 Conventional Processors ................................... 15  
2.1.2 SMPs, MPPs, and Beowulfs .................................. 17  
2.2 Alternative Hardware ........................................... 18  
2.2.1 PIMs ............................................................. 18  
2.2.2 Vectors .......................................................... 19  
2.2.3 Multithreading ................................................ 19
4.5.1 Vectors Improve Matching .................................. 105
4.5.2 Effects on Throughput .................................. 107
4.5.3 Effects on Latency .................................. 108
4.5.4 Conclusions .................................. 112

CHAPTER 5: FULL/EMPTY BIT PIPELINING ................. 114
5.1 FEB Pipelining Design .................................. 118
5.2 FEB Pipelining Implementation .......................... 119
5.3 Experimental Setup .................................. 122
   5.3.1 Simulated Hardware ................................. 122
   5.3.2 Benchmarks And Applications ..................... 124
   5.3.3 Analysis .................................. 125
5.4 Results .................................. 126
   5.4.1 FEB Pipelining Accelerates Application: Micobenchmark 126
   5.4.2 FEB Pipelining Accelerates Application: NAS Subset .. 129
   5.4.3 FEB Pipelining Accelerates Application: Analysis .... 131
5.5 Conclusion .................................. 132

CHAPTER 6: LOOP LEVEL THREADING ................. 133
6.1 Enabling Threads .................................. 135
6.2 Design .................................. 137
   6.2.1 Hardware .................................. 138
   6.2.2 Application Software ............................. 139
6.3 Implementation .................................. 139
   6.3.1 Hardware Implementation ......................... 139
   6.3.2 OpenMP on pthreads Implementation ............. 141
   6.3.3 Work Scheduling ................................ 142
      6.3.3.1 OpenMP Limits ................................ 142
      6.3.3.2 Heterogeneous Scheduling ..................... 144
   6.3.4 Data Management ................................ 144
      6.3.4.1 Software Cache ................................ 145
      6.3.4.2 Migration .................................. 146
      6.3.4.3 Stack Cache ................................ 147
6.4 Experimental Setup .................................. 147
   6.4.1 Simulated Hardware ................................. 147
   6.4.2 Benchmarks .................................. 149
6.5 Results .................................. 150
   6.5.1 Branch Instruction Distribution ................... 150
   6.5.2 Cache Pressure .................................. 152
   6.5.3 Silicon Cost/Performance ......................... 153
   6.5.4 Data Management ................................ 156
   6.5.5 Conclusion .................................. 158
## FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Simplified conventional system</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Proposed Light Weight Processor organization</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Simulator Framework and Components</td>
<td>57</td>
</tr>
<tr>
<td>3.2</td>
<td>Enkidu pseudocode</td>
<td>64</td>
</tr>
<tr>
<td>3.3</td>
<td>Simulator front ends</td>
<td>66</td>
</tr>
<tr>
<td>3.4</td>
<td>Some back end component classes and their relationships</td>
<td>69</td>
</tr>
<tr>
<td>3.5</td>
<td>Processor/Thread Interface</td>
<td>70</td>
</tr>
<tr>
<td>3.6</td>
<td>Memory system components</td>
<td>73</td>
</tr>
<tr>
<td>3.7</td>
<td>2-D mesh router</td>
<td>75</td>
</tr>
<tr>
<td>3.8</td>
<td>Sample two node system</td>
<td>77</td>
</tr>
<tr>
<td>3.9</td>
<td>Simulator Comparison</td>
<td>79</td>
</tr>
<tr>
<td>4.1</td>
<td>High Performance NICs</td>
<td>85</td>
</tr>
<tr>
<td>4.2</td>
<td>Subset of MPI implemented</td>
<td>88</td>
</tr>
<tr>
<td>4.3</td>
<td>Conventional and PIM data layouts</td>
<td>92</td>
</tr>
<tr>
<td>4.4</td>
<td>Vectorized Comparison Process</td>
<td>95</td>
</tr>
<tr>
<td>4.5</td>
<td>Comparison instructions vs. vector size</td>
<td>100</td>
</tr>
<tr>
<td>4.6</td>
<td>Pseudocode for preposted queue impact</td>
<td>103</td>
</tr>
<tr>
<td>4.7</td>
<td>Message timelines without vectorized matching</td>
<td>104</td>
</tr>
<tr>
<td>4.8</td>
<td>Message timelines for receive-side on LWP with vectorized matching</td>
<td>105</td>
</tr>
<tr>
<td>4.9</td>
<td>Message timelines for receive-side on conventional NIC</td>
<td>107</td>
</tr>
<tr>
<td>4.10</td>
<td>Conventional NIC bandwidth curves</td>
<td>109</td>
</tr>
<tr>
<td>4.11</td>
<td>Conventional NIC latency curves</td>
<td>110</td>
</tr>
<tr>
<td>4.12</td>
<td>Receive-side timing breakdowns</td>
<td>111</td>
</tr>
<tr>
<td>5.1</td>
<td>Typical application communication</td>
<td>116</td>
</tr>
</tbody>
</table>
TABLES

1.1 FOCUS OF EFFORT: ARCHITECTURAL INNOVATIONS . . . 12
2.1 FEB MANIPULATION INSTRUCTIONS . . . . . . . . . . . . . 28
2.2 COMPARISON OF LWP AND NIAGARA CORES . . . . . 29
2.3 KEY MPI DATA STRUCTURES . . . . . . . . . . . . . . . . . . . 48
3.1 KEY COMPONENTS . . . . . . . . . . . . . . . . . . . . . . . . 72
4.1 LWP CORE SIZES . . . . . . . . . . . . . . . . . . . . . . . . . . 86
4.2 ALTIVEC SUBSET . . . . . . . . . . . . . . . . . . . . . . . . . . 96
4.3 INSTRUCTION COSTS FOR VECTORIZED COMPARISON . . 97
4.4 COMPARISON INSTRUCTIONS VS. VECTOR SIZE . . . . . . 99
4.5 PROCESSOR SIMULATION PARAMETERS . . . . . . . . . . . 101
5.1 SIMULATOR PARAMETERS . . . . . . . . . . . . . . . . . . . . 123
5.2 APPLICATION ANALYSIS . . . . . . . . . . . . . . . . . . . . . 131
6.1 NAIVE LOOP THREADING RESULTS . . . . . . . . . . . . . . 137
6.2 SOFTWARE CACHE SIZES AND USAGE . . . . . . . . . . . 146
6.3 PROCESSOR SIMULATION PARAMETERS . . . . . . . . . . . . 148
6.4 PRICE PERFORMANCE GRAPH EXPLANATION . . . . . . . 155
7.1 LEVELS OF PARALLELISM . . . . . . . . . . . . . . . . . . . . 161
7.2 MEAN THREADLET PERFORMANCE (IPC) . . . . . . . . . . 180
8.1 PERFORMANCE IMPACT OF DECREASED OVERHEAD . . . . 188
8.2 IMPACT OF FEB PIPELINE . . . . . . . . . . . . . . . . . . . . 189
8.3 ESTIMATED CUMULATIVE EFFECT ON PERFORMANCE . . . 191
8.4 LWP FEATURE USAGE . . . . . . . . . . . . . . . . . . . . . . . 192
8.5 USES OF A RUNTIME MONITOR ......................... 200
8.6 POTENTIAL DESIGN PATTERNS ....................... 201
ACKNOWLEDGMENTS

Parts of this material is based on a collaboration with Sandia National Labs. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000. Specific work was carried out under contracts #005934-001, #384693, and #478022.

This material is based in part upon work supported by the Defense Advanced Research Projects Agency (DARPA) under its Contract No. NBCH3039003.

I would like to give special thanks to my advisor, Dr. Peter Kogge for allowing me to pursue these projects. His abilities to guide (or not guide) as required and to continually confound and baffle me (as required) have been invaluable.

The feedback from my committee was invaluable to making this document successful.

Finally, this work could not have been accomplished without continual assistance from my friends and colleagues. I would particularly like to thank:

- Elizabeth White for her network models,
- Kyle Rupnow for his application analysis,
- Shyamkumar Thoziyoor for his sizing estimation tools,
- Jeff Squyres, Brian Barrett, and Ron Brightwell for explanations of how MPI really works,
• Kyle Wheeler for forcing me to think “like an OS guy” from time to time,

• Richard Murphy, for vexing me at every turn,

• Keith Underwood for his indispensable supervision, direction, and for teaching me that “Everything proven by experience elsewhere fails in New Mexico,”

• Srinivas Sridharan for making me write things down.
CHAPTER 1

INTRODUCTION

High performance scientific computing\(^1\) has been powered by two trends: increasingly powerful hardware and increasingly capable software. Due largely to advances in CMOS fabrication technology\([96]\), the processing, storage, and communication capabilities of processors, memory, and networks have continually grown since the mid 1970s \([134]\). This growth has fueled new software which can tackle increasingly large, complex, and detailed problems. In addition, software has changed by adding new capabilities. Entirely new classes of software have emerged to utilize graphical user interfaces, visualization capabilities, and network functionality.

However, the meeting point of these hardware and software trends may become a bottleneck to both. This meeting point, the programming model, has remained surprisingly static when compared to the rapid evolution of hardware capabilities and changing requirements of software. Commonly used scientific benchmarks such as the NAS Parallel Benchmarks \([213]\) or ASCI Purple \([157, 186]\) and production workloads \([154–156, 166, 210]\) are comprised almost entirely of programs written in FORTRAN \([48]\) or C \([101]\) with C++ \([195]\) accounting for the rest\([185]\).

---

\(^1\)For the purposes of this document, high performance scientific computing refers to parallel scientific simulation.
Many existing production codes are the result of decades of man-years of development and there is a strong disincentive to radically rewrite these “Dusty Decks.”.

These long-lived applications and programming models build upon themselves. Programs created in C, C++, or FORTRAN must be maintained in C, C++, or FORTRAN. This requires the hiring of programmers versed in these languages. This in turn creates a demand on the educational system to produce programmers trained in these languages. Because programmers tend to write in languages they are familiar with, this ensures that new applications are written in old languages.

Entrenched ‘legacy’ codes have placed a significant burden on hardware. New hardware must remain compatible with existing software to be economically viable. Significantly new hardware capabilities must not interfere with the performance of legacy software, ideally being completely hidden from the high level programming language. A nonconventional hardware capability can only justify its cost if there is a significant increase in legacy performance.

The legacy requirement has traditionally limited the ability of hardware to take advantage of novel architectural features such as processing-in-memory, vector processing, multithreading, or dataflow designs. As a result, supercomputing is increasingly dominated by clusters of commodity scalar processors which do not support novel architectural features[202]. In turn, this has limited the feasibility of new programming languages and models such as Split-C, UPC, SISAL, or parallelizing compilers which would benefit greatly from hardware support.

Though very successful, the standard cluster of scalar processors faces increasing challenges. Many of these challenges revolve around the growing disparity between processor and memory speed [215] - the “Memory Wall.” In addition, scaling applications to efficiently use the large number of parallel processors re-
quired to achieve high performance is becoming more difficult because of network limitations and difficulty in decomposing problems.

To a large extent, these limitations are inherent to the traditional computer architecture comprised of a processor, memory, and bus. As data is consumed in computation more quickly, marshaling that data to the point of computation is more difficult. Barring any upward revision in the speed of light, this will continue to be a fundamental limitation on the speed of computation.

To provide the capabilities demanded for future generations of high performance scientific computing, a new system must provide two things. First, novel hardware architectures must form the foundation for increased performance. Secondly, tools and techniques must be made available to transform legacy applications into a form which can efficiently run on these novel architectures and enable these applications to use the hardware’s new capabilities.

1.1 The Overall Problem

The problems facing supercomputing fall into two broad categories. Hardware problems which prevent scalability, and software problems which inhibit the adoption of new techniques and technologies.

1.1.1 Hardware: Diminishing Returns

To some extent, hardware is a victim of its own success. The great advances in VLSI[159] design and CMOS fabrication have provided a continual source of growing raw performance. Architectural advances such as deep pipelining[82, 88, 187], superscalar processors [183], caches [83], and branch prediction made computers “fast enough” without requiring much intervention by the programmer. Tech-
niques which would require modification of the software, such as multithreading, were generally not seen as worth the cost.

For example, the out-of-order instruction execution in most modern processors had little effect on the programming model, but yielded a performance boost large enough to justify the cost. A similar example: vector processing, though once used in over 60% of the top supercomputers [202], fell by the wayside by the mid 1990s and was largely replaced by cheaper commodity scalar processors which did not require code changes to utilize vectorization. Currently, less than 7% of the top supercomputers are vector based.

These successes cannot continue indefinitely. The growing “memory wall” already demands extremely complex latency avoidance techniques such as multi-level caches and prefetching. Parallelism, within a single processor, is currently addressed by out-of-order superscalar execution. Increasing this parallelism by adding more functional units yields diminishing returns and comes at the cost of exponentially increasing power consumption [72, 163]. Additionally, network bandwidth and latency concerns may pose significant hurdles to scalability.

1.1.2 Software: Productivity, or Programming is Hard

Unlike the natural languages in which human-to-human communication is conducted, computer languages are grammatically simple, syntactically rigid, and unambiguous. In spite of these constraints (or perhaps because of them), “speaking” a computer language is not as unconsciously natural an action as speaking a natural language [152]. Instead, our thoughts must be translated or “distilled” down to the very limited subset offered by computer languages. This translation process, unaided by the buffer of ambiguity provided by natural language, is time consuming and prone to error. This is the crux of the productivity problem.
Though progress has been made, attempts to “solve” this problem have met with limited success. New languages and programming models which attempt to vastly improve the situation run across several difficulties:

- **Inherent Complexity**: Writing useful scientific programs is a difficult and complex task. This is to be expected, as most useful science is difficult and complex. Though a wide range of tools, languages, libraries, and design patterns exist to make programming easy, there is no ‘silver bullet’ [24] which makes programming a universally simple task.

- **Maintainability and debugging** Program maintainance and debugging are major hurdles for new languages. Often, tools for performance profiling and debugging are not mature on new languages, which can put them at a significant disadvantage. New languages can also present a steep learning curve, especially if they adopt a radically new programming model. This can lead to a smaller pool of programmers familiar with a new language and can make finding maintainers more difficult. For these reasons, new languages and the new features they showcase are often greeted with mistrust.

- **Portability** The rapidly changing field of high performance computing makes portability a requirement for most applications. Only 3 of the current top 50 supercomputers were added before 2001, indicating a high turnover rate for computing resources. This high turnover requires applications to be portable. This portability requirement makes it difficult to take advantage of new features in a language or architecture.

Because of these obstacles, new languages and models are slow to emerge and applications tend to be written in “legacy” languages. The inertia of the existing
codebase requires that new architectures be able to support, and even accelerate, these “dusty decks.”

1.2 Community Goals

The scientific computing community must address these problems to provide the capabilities required for future computational demands. To properly address the issues, we must ask what future generations of supercomputers will look like, and how we expect to program them.

1.2.1 What Should the Next-Generation Supercomputer Look Like?

This work focuses on the “next-generation” supercomputer. We assume that this machine is one which will become available sometime around 2010. We assume it will be based on CMOS transistor technology rather than a novel post-transistor technology such as QCA[115]. It is safe to assume that this machine will be massively parallel, with the equivalent raw performance of several tens or hundreds of thousands of conventional commodity processors.

Many hardware technologies are being considered for such a machine. Vector processing, large shared memory, radically reorganized memory hierarchies, streaming [162], SMT, processing-in-memory, and advanced networking technology are all likely contenders for future supercomputers. However, equally likely is that the next-next generation of supercomputers will be essentially the same as today but scaled up.

Due to the conservative nature of hardware development imposed by legacy software, the most successful new hardware will require minimal changes to existing programming methods. Additionally, the added cost of any new hardware
system will have to be minimal to compete with the dominant commodity cluster approach of today.

1.2.2 What Should the Next-Next-Generation Supercomputer Look Like?

It appears that conventional CMOS technology may “run out of steam” sometime in the next 10 to 25 years. This is indicated by the growing number of production hurdles in the ITRS roadmap[96] which have no known solution. A number of technologies are contending to replace CMOS. Additionally, the programming models and applications of this time frame are difficult to predict. Overall, the shape of a computer in 2030 is far from clear.

Though the community cannot answer this question fully, we can deduce some qualities about this future computer. It can be reasonably assumed that the next-next-generation of high performance computers will exhibit a much higher degree of parallelism and require more memory resources than current machines. Additionally, the fundamental barrier of the speed of light will mean that locality will still be an issue. Mechanisms will have to exist to coordinate the flow of data and scheduling of computation in a timely manner.

1.2.3 How Should We Program It?

When asking what the proper programming model of the future is, we must first ask what the programming model(s) of today are, and why they should they change. If current models cannot scale to larger problem sizes or they cannot take advantage of new architectures, what are the features which must be added?

Finally, and most critically, is it worth the change? Would the potential benefits of a new language or programming model outweigh the cost in creating new
tools, learning new models, and writing new code? Or, would it be more efficient (though less elegant) to “brute force” our current models onto the machines of tomorrow? Due to the massive investment in current legacy applications, any new model would have to provide enormous and lasting benefits to be economically feasible.

1.3 Dissertation Goals

This work focuses on these community goals in the context of Light Weight Processing (LWP). The LWP “philosophy” is to move the site of the data and the computation closer together, and to exploit parallelism by breaking down computation into smaller units executing on an abundance of simple processors. Specifically, the LWP explored here is a combination of processing-in-memory[21] and hardware multithreading which also leverages short vector processing and extended memory semantics for low level synchronization.

This work is most directly related to resolving the “next-generation supercomputer” question. However, the fundamental techniques it demonstrates will remain applicable to more advanced systems.

This work does not attempt to “solve” the fundamental problems of programming productivity. Rather than try and construct a new language to express computation, it aims to identify generic techniques and patterns – “metaphors” – for using LWP to address high performance computing. Like metaphors in language, these techniques are language neutral.

---

Footnotes:

2Throughout this work Light Weight Processing will be abbreviated as LWP. LWP will also be used to identify Light Weight Processors as defined in section 2.2.6.

3To paraphrase Shakespeare, What’s in a language? That which we call an MPI_Recv(), By any other calling convention would smell as sweet.
The goal of this work is to demonstrate the feasibility of LWP-based supercomputers. To be feasible, LWP-based computers must

- execute “dusty deck” programming models efficiently,
- provide improved performance by circumventing the “memory wall”,
- and show architectural features which will address the fundamental long-term issues in high performance programming (such as increased parallelism and locality issues).

To this end, this work addresses two major targets areas:

1.3.1 MPI: The Now and Future King

Applications of LWP to the Message Passing Interface (MPI) are examined. MPI is widely regarded as the de facto standard for scientific computing. Its continued dominance, assured by billions of dollars of legacy code and widespread support by business, government, and academia, will almost certainly continue into the next generation of supercomputers. Today’s MPI is the dusty deck of tomorrow.

LWP techniques can be applied to improve MPI in a number of ways:

- The processing of messages within the “MPI engine” can be accelerated by using multithreading and closer proximity to memory to accelerate MPI’s internal mechanics.
- The extended memory semantics made possible by LWP open up new possibilities for overlapping computation and communication. This allows increased performance without changing the MPI syntax and may simplify programming.
1.3.2 Relentless Parallelism: Finding Threads

LWP relies on a large number of “cheap” processors performing work in parallel. To accomplish this, it is necessary to extract large numbers of threads from an application.

This extraction can be confronted on two separate but complimentary levels:

- LWP can be integrated into a conventional architecture by augmenting the existing CPU. To facilitate this, the computation must be broken down into smaller threads of execution and work must be distributed between the conventional processor and the LWPs. This is explored by using a modified OpenMP compiler and library to identify threads and distribute them between multiple processors.

- The simplicity of the LWP’s execution core can make single thread performance poor. Conventional processors use complex superscalar techniques to improve performance, but at a considerable cost in chip area. A better approach for LWPs is to move the burden onto the compiler by extracting very small threads, or “threadlets” from basic blocks.

1.4 Approach: Parallelism in New Places

At a high level, both of these targets follow a similar broad approach. The recurrent theme is to find parallelism in new places. This approach stems from the LWP philosophy of using many simple processors in parallel. In economic terms, abundance of processing sites increases the demand for threads. To meet this demand, new supplies must be tapped.
The approach used in this work is to try and extract threads at multiple levels of parallelism. At the process level, improving MPI and overlapping computation and communication is used. At the loop-level and function level, using OpenMP to find threads and balance them between processors. And, at the basic-block level, extracting and executing threadlets allows very fine grained parallelism to be addressed.

1.5 Scope

The scope of this work is defined by its discipline, approach, and effort. That is, the set of problems it addresses, the set of techniques it uses, and the set of problems it emphasizes form the boundaries of the scope of this work.

The discipline is high performance scientific computing (HPC). This includes the use of computers to solve scientific or engineering problems. This differentiates the scope from other subsets of computing such as desktop applications, real-time or embedded applications, and gaming. Additionally, this work is limited to high performance computing, where the speed at which a solution is reached is of critical importance. This differentiates the work from scientific computing which may be more exploratory in nature, but not require high performance.

The approach used generally in HPC is a combination of three aspects: algorithm, science, and architecture. The algorithm, in the broadest sense, defines how information is processed and analyzed to come to a solution. For this work, we assume traditional algorithms which represent data as binary numbers and perform mathematical transformations on these numbers in an iterative manner. This differs from other emerging techniques such as quantum computing or DNA-


<table>
<thead>
<tr>
<th>FOCUS OF EFFORT: ARCHITECTURAL INNOVATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bandwidth</strong></td>
</tr>
<tr>
<td>Communication</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>Computation</td>
</tr>
<tr>
<td>I/O</td>
</tr>
</tbody>
</table>

based computing. The science which underlies these traditional computations is assumed to be some form of simulation of a mechanical deterministic system governed by mathematically expressed equations. Again, emerging alternatives, such as artificial intelligence, are excluded. The primary aspect explored in this work will be the architecture used to support the algorithms and underlying science.

This work places its effort into a subset of architectural innovations. Broadly speaking, improvements can be categorized on a four by three matrix (see Table 1.5). On one axis of the matrix is the area to be improved: communication, memory, computation, and I/O. Improvements to each of these can be brought about in one of three ways: increased bandwidth, improved latency, or improved process. For example, communication can be improved with a “wider” network connection (bandwidth), lowered overhead through a different network protocol (latency), and fundamental changes through a new underlying process, such as an optical network. Similarly, computation has been improved by making processors “wider” through multiple pipelines (bandwidth), masking latency with pipelining.
(latency), and increased clock rates and densities through fabrication improvements (process). The scope of this work is to address innovation by improving bandwidth and latency of communication, memory, and computation. We neglect the effects of fundamental process improvements and also the effects of I/O.

All of this is not to say that the techniques presented in this work can not find application beyond this scope. However, to maintain focus, this work will be bound by the scope presented above.

Because of these limitations to scope, this work cannot prove the optimality of an LWP-based system for all computing. However, it is clear that communication, memory performance, and computation will remain important aspects of computing, and a major aspect of addressing these concerns will be to improve their latency and bandwidth. What this work does show is that LWP addresses these issues on a fundamental level.

1.6 Thesis Contributions

This work presents several contributions which demonstrate the effectiveness of LWP-based computing:

- An implementation of MPI which uses features of LWP to substantially improve message processing throughput.

- A technique leveraging the low-level synchronization mechanisms of LWPs to organize tight producer-consumer relationships between MPI processes and improve overlap between computation and communication.

- Modifications to an OpenMP library and LWP run-time support to allow efficient partitioning of threads between a conventional CPU and LWPs.
This partitioning can improve performance and greatly improve cost/performance.

- An algorithm to identify and extract very small “threadlets” which can overcome the inherent disadvantages of a simple processor pipeline.

1.7 Roadmap

Chapter 2 covers the state of the art of both high performance hardware and software.

Chapter 3 goes over the Structural Simulation Toolkit—a simulation framework used in much of this work.

Chapter 4 shows an MPI implementation which uses features of LWP to improve message processing throughput.

Chapter 5 introduces a technique which leverages the extended memory semantics of LWP to produce producer-consumer relationships between processes and improve communication.

Chapter 6 explains modifications to an OpenMP library to allow efficient partitioning of threads between a conventional CPU and LWPs—greatly improving cost/performance.

Chapter 7 shows an algorithm to extract very small “threadlets” which can overcome the inherent disadvantages of a simple processor pipeline, such as is found in an LWP.

Lastly, Chapter 8 concludes the work with a summary and possible future enhancements and extensions.
Currently, most high performance computing occurs within a well defined set of conventional hardware and software. Section 2.1 covers the conventional hardware used today. Sections 2.3 and 2.4 examine the prevalent software, with special attention to the Message Passing Interface (MPI) which is the de facto standard for parallel programming.

On the periphery are other emerging technologies and trends which seek to augment or displace the incumbents. Section 2.2 looks at emerging hardware trends which offer a substantial shift in how computing is performed. Section 2.5 examines software technologies and alternative programming models which have tried to redefine how parallel programs are constructed.

2.1 Conventional Hardware

Most hardware tends be organized in a configuration similar to that shown in Figure 2.1. A CPU is connected, through one or more levels of fast cache, to a main memory. The main memory is comprised of banks of DRAM chips.

2.1.1 Conventional Processors

Modern CPUs are generally superscalar, out-of-order processors. A typical multiple issue processor fetches several instructions in a single clock cycle. These
Figure 2.1. Simplified conventional system
instructions are analyzed for data dependencies, some of which are eliminated through register renaming. The instructions are then distributed between multiple functional units and are executed as the required data becomes available. Results from completed instructions are “committed” to a copy of processor state which reflects in-order execution. This copy can be used to retrieve a “safe” version of processor state in the event of an exception or interrupt [163, 183]. Each instruction is generally a scalar instruction. Floating point operations are usually handled by a specialized functional unit.

2.1.2 SMPs, MPPs, and Beowulfs

The single processor system organization shown in Figure 2.1 is modified for parallel systems. Symmetric Multi-Processing (SMP) systems such as the Sun Enterprise Server [130] or Apple Power Mac G5[35] tightly couple multiple processors into the same system. These processors all use the same main memory, though usually retain separate caches. Parallelism in SMP systems is achieved by using multiple threads within the same process and allowing the operating system to distribute these threads between the available processors, or by having multiple processes executing and using shared memory, MPI, or some other mechanism for interprocess communication.

To scale to larger systems, the coupling between processors is reduced, leading to Massively Parallel Processing (MPP) architectures or Beowulf[173] clusters. Here, large numbers of processors, each with their own main memory, are connected with a network. Generally, these processors have their own completely separate address space. Beowulf clusters focus on using commodity off-the-shelf (COTS) hardware as much as possible to reduce cost. However, as high-speed low-
latency networks such as Myrinet[14, 36] drop in price, the dividing line between MPP and Beowulf blurs.

2.2 Alternative Hardware

To challenge the conventional system and processor organization, a number of technologies are appearing.

2.2.1 PIMs

One alternative organization is to merge processing elements and main memory data storage onto the same physical chip die. This approach, Processing-In-Memory (PIM), also called Intelligent RAM[149], embedded RAM, or merged logic and memory, has been explored in a number of projects [51, 52, 54–57, 59, 61, 103, 104, 132, 140, 146, 191–193]. These projects have explored programming and execution models, microarchitectural organization, and physical design and layout.

These have covered a range of execution semantics and programming models [22, 105, 106, 141]. On the system level, they have been looked at as memory-based accelerators to existing processors (DIVA[79]) or as the sole processing elements for a stand alone (usually MPP) machine[3, 107].

The chief advantages of combining processing and memory are:

- **Lower Latency**: The proximity of memory allows fast access to data. Memory access speeds of less than 10ns are considered possible.

- **Greater bandwidth**: Because the transmission distance is lower, it is possible to leverage large amounts of on-chip bandwidth by connecting processing elements directly to memory macros.
• **Less caching**: The lower latency and bandwidth may reduce the need for caches, which eliminates a number of issues with cache coherency.

2.2.2 Vectors

Vector processing gained popularity in the mid-1970s with the introduction of the CRAY-1[171]. By supporting long vectors and iteration through vectors with non-unit stride, vector processing was applicable to a large number of scientific problems.

However, by the mid 1990s, custom designed vector processors were being supplanted by inexpensive commodity scalar processors. These commodity processors could benefit from economies of scale provided by the vast consumer and business PC market.

Interestingly, it is the home consumer PC market which has brought a resurgence to a limited form of vector processing. Short vectors, which focus on smaller vectors of unit stride, are commonly used in the graphics processing for consumer games and multimedia applications. This has lead to extensions to scalar ISAs such as Intel’s MMX[150] and the PowerPC Altivec[142] to provide short vector support.

2.2.3 Multithreading

Hardware support for multiple threads\(^1\) has been appearing in a growing number of commodity processors [15, 109, 116, 172], often in an attempt to boost throughput for server workloads [126]. Multithreading has also been used by the

---

\(^1\)Though terminology differs, for this work “thread” is defined as a sequence of executing instructions which may reside in the same address space as other such threads. This distinguishes it from a “process” which has a separate address space and is generally “heavier weight.”
Cray (Tera) MTA[5] to parallelize a wide variety of scientific applications and reduce the effects of memory latency. Additionally, multithreading has been used in a variety of network processors such as the Intel IXP family or Cisco Toaster[81].

The chief benefits of multithreading:

- **Parallelism**: Breaking a problem into multiple threads allows those threads to be executed in parallel on multiple processors.

- **Latency Tolerance**: Latency from long operations such as memory, vector, or floating point computation can be tolerated by breaking a computation into threads. The latency from threads which are consumed by high latency activities can be masked by continuing to execute low latency threads. For example, a common use of threads is to have one thread which deals with the user interface of a program while another thread performs the actual computation.

- **Productivity**: A number of programming problems naturally decompose into a multithreaded solution. For example, often networking applications with a client/server architecture can use multiple server threads to handle multiple clients. Also, thread-level software pipelining can be used to separate logically distinct tasks into multiple threads.

Adding hardware support for multithreading reduces the overhead with thread creation and synchronization. This allows threads to be applied to a larger number of problems and with greater efficiency.

Hardware synchronization support has a long history in research and production systems[2, 5, 20, 47, 148, 196, 200]. The Denelcor HEP [47], introduced in 1982, used basic hardware support to provide simple mutual exclusion. The Hori-
zon machine [200] in 1988 had similar memory semantics. The Monsoon dataflow
computer [148], introduced two years later, used a modified form of full-empty bits
in the form of “presence bits” which were used to indicate whether an operand has
been generated for a memory address. More recent implementations of full-empty
memory semantics have been in architectures that focus on lightweight threads
and large amounts of parallelism, such as the PIM Lite project [20] and IBM’s
CELL processor [196]. Other novel synchronization techniques include allowing
adjacent processors to share register sets (used in the Intel IXP processors).

2.2.4 Multicore (CMP)

Another popular trend has been to place multiple processor cores on a single
chip die. These Chip Multiprocessor (CMP) systems offer increased parallelism
with a lower overhead and faster core-to-core communication than a traditional
SMP.

This technique is used in a number of specialized processors for gaming [196]
and network control [80, 216]. It has also seen growing use in general purpose
processors. In the 2006 IEEE International Solid-State Circuits Conference all
five of the newly introduced processor designs used multiple cores. Most of these
designs were dual core [34, 70, 172] where the cores were essentially the same as
previous single core offerings. Of special note is the SPARC Niagra Processor [116]
which uses eight cores to support up to 32 threads. However, instead of replicating
a traditional conventional superscalar core eight times, this chip uses simplified
cores which reduce power dissipation [109]. A similar approach is taken in IBM’s
Cyclops architecture [52, 53] which also uses multithreading and multiple simple
cores.
2.2.5 Dataflow Machines

Dataflow machines [208] are a class of computers where the hardware is optimized for fine-grain data-driven computation. As opposed to conventional control-flow computation, data-driven computation operations are scheduled on the basis of the availability of their operands. In a sense, this is multithreading taken to the extreme, where each instruction becomes a thread.

2.2.5.1 Tagged Token Dataflow

In tagged-token dataflow machines, such as MIT’s Tagged-Token Dataflow Architecture, the Manchester Prototype Dataflow Computer[77], and SIGMA-1, data values are “tagged” with a unique identifier. These values are transmitted in packets known as “tokens” (hence, tagged-token). A complex matching unit pairs tokens destined for the same instruction[148]. When all the tokens required to start a given instruction are matched, that instruction begins execution. Upon completion, the instruction may generate additional tokens which will start more instructions. Variants of this method, such as the Explicit Token Store Architecture (ETS) used in the Monsoon computer[148], reduce the token matching overhead by providing explicit locations for synchronization to occur.

2.2.5.2 Hybrid Dataflow

Even with the explicit coordination of the ETS architecture, the overhead of matching every token can be considerable. An alternative is to hybridize pure dataflow and traditional control-flow computing[91]. In this model, it is not necessary to tag and match every data value and instruction. Instead, hybrid models, such as P-RISC[143], use static control-flow scheduled code when it is profitable to
do so, but use data-directed synchronization to coordinate parallel execution[169]. This allows lower overhead execution of serial sections while still gaining performance from parallelism. This technique is potentially general enough to enable dataflow execution without hardware support, such as the TAM (see section 2.5.5).

2.2.5.3 Dataflow Problems

The question then becomes, why did data flow machines fail?

Three possible answers for this:

1. **They Didn’t**: Though dataflow as a programming model did not become widely accepted at the application level, the tag matching system is the basis for the out of order execution model used in most modern processors[83].

2. **Economic**: Dataflow was not compatible with much existing software, therefore limiting its acceptance. Thus, dataflow machines could not benefit from the economies of scale available to conventional commodity processors.

3. **Productivity**: To take maximum effect of dataflow techniques, specialized dataflow languages, such as SISAL[131], were required. These languages lacked a user base and did not provide enough incentive to adopt. In particular, loops were difficult to specify.

Most likely, answers 2 and 3 combined to present a “chicken and egg” scenario - dataflow lacked the hardware performance to justify adopting new programming models, and dataflow hardware lacked performance because it didn’t have the user base to justify economies of scale to fabricate high performance chips.

Answer 1 may provide a lesson for other architectural innovators - a radical hardware organization can gain widespread acceptance, but only if it requires no
changes to the programming model and retains full compatibility with existing code.

2.2.6 LWPs

This work focuses on an emerging hardware paradigm called Light Weight Processing (LWP). LWP is a combination of several emerging techniques. Chiefly, it uses Processing-in-Memory and multithreading. It extends and leverages these techniques in the following ways:

- **Short Vector**: The proximity of memory and increased bandwidth provided by processing-in-memory may allow the LWP to perform short vector operations more easily. This takes advantage of the increased width of data found at the memory macro interface.

- **Extended Memory Semantics**: The close relationship with memory also allows memory to be “tagged” with meta-data which can be used for thread synchronization. For example, each word in memory could contain an extra bit indicated if that word is “full” or “empty”. This Full/Empty Bit (FEB) could be accessed by threads to determine if the word contains valid data. This could be used to set up a producer/consumer relationship between threads[1, 204]. The same technique could even be applied to words in a register file to allow several small threads to share resources [160]. Further, placing the memory state processing close to the memory interface greatly reduces the latency of processing that state.

- **Communication Capabilities**: Because the amount of memory on a single die is not sufficient to support most scientific computing problems, LWPs
will have to be closely tied to an efficient communication mechanism, such as parcels (PARallel Computing Elements)[22].

2.2.6.1 A Proposed LWP

One proposed organization for an LWPs is shown in figure 2.2. This organization evolved from proposals related to the Cascade project[179, 182]. In the proposed organization, several Light Weight Processors are placed on a single die, a Light-Weight Processing Chip, or LPC. These processors (LWP Pipe) are simple, short, interwoven, threaded pipelines. Each LWP has access to a Frame Cache, which contains the register sets for a number of threads currently pending execution. The pipelines will have access to an SRAM instruction cache (iCache) which may be shared by other pipelines. An interface to the outside world is provided by the Parcel Handler which receives and assembles incoming messages and sends outgoing. The LWPs are connected to each other and to the Parcel Handler by a network (SW1). This network also connects the LWPs to embedded DRAM (EDRAM) which is also on-chip. This embedded DRAM is organized like a cache. There may be a separate EDRAM for instruction data. Another network (SW2) connects the LPC chip to backing DRAM chips. The LPC may be connected to other LPCs, or there may be a conventional “heavyweight” processor directing computation.

2.2.6.2 Thread Scheduling

Thread scheduling uses a simple mechanism. Threads are initially placed in a frame cache, a hardware structure that contains the thread state for a number of threads. Threads in the frame cache can be executed immediately (assuming the
Figure 2.2. Proposed Light Weight Processor organization
thread is not blocked). In normal operation, the scheduler issues from threads in the frame cache in a simple round-robin fashion.

However, because the frame cache is of finite size, an additional thread storage area is assumed to exist in main memory. If there are enough threads to “overflow” the frame cache, their state will be placed in memory. If there are threads in memory, every 256 cycles the scheduler will stop execution and try to swap threads between the frame cache and memory. This swapping is done in a FIFO manner so threads which have been in the frame cache will be swapped out in favor of threads which have been dormant in main memory the longest.

2.2.6.3 ISA Modifications

For the experiments in this work, the LWP used the PowerPC ISA. To support additional features of the LWP architecture, some changes were made to the baseline PPC ISA.

To support short vector operations (Chapter 4), a subset of the Altivec vector ISA was added. These changes are detailed in Section 4.3.3.2.

The majority of changes were included to support changes in thread handling and to support Full/Empty Bit (FEB) semantics. To create and destroy threads, the \texttt{PIM\_FORK} and \texttt{PIM\_EXIT} instructions were added. Note, the \texttt{PIM\_FORK} instruction creates a child thread whose registers are a copy of its “parent,” except for register \texttt{r3} (the argument register in the PPC conventions) and the program counter. \texttt{PIM\_EXIT} simply ends the thread’s execution, so it is up to the user or thread library (such as \texttt{pthread}) to ensure that any return value from the thread or other cleanup is performed. Additionally, several variants of load and store were created to allow manipulation of the FEBs. These instructions are summarized in
TABLE 2.1

FEB MANIPULATION INSTRUCTIONS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_ff</td>
<td>Read a given word if its FEB is “full,” otherwise, block. After the read, the FEB remains “full.”</td>
</tr>
<tr>
<td>read_fe</td>
<td>Read a given word if its FEB is “full,” otherwise, block. After the read, the FEB is set “empty.”</td>
</tr>
<tr>
<td>change_fe</td>
<td>Change the FEB associated with a given word to “full” or “empty.”</td>
</tr>
<tr>
<td>write_ef</td>
<td>Write a word to memory at a given address if that address is “empty.” Afterwards, set the address’s FEB to “full.”</td>
</tr>
<tr>
<td>try_ef</td>
<td>If a given word is “empty,” set it to “full” and return 0. If the word is “full” return 1.</td>
</tr>
</tbody>
</table>

Table 2.1. Of note is the try_ef instruction, which is modeled after the pthread library’s mutex_trylock() function.

2.2.6.4 Memory Accesses

The physical memory space is segmented such that each LWP has exclusive access to a range of addresses. An LWP is said to “own” the physical addresses in its range. Any LWP can access the entire address range, however, if it requires memory which is “owned” by another LWP, it must issue a remote instruction to the owning processor. The remote LWP fetches the requested memory and returns the data to the requesting LWP. During this time, the thread which has
TABLE 2.2

COMPARISON OF LWP AND NIAGARA CORES

<table>
<thead>
<tr>
<th>Parameter</th>
<th>LWP</th>
<th>Niagara</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline Stages</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Available Threads</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>L1 iCache</td>
<td>4KB/8-Way</td>
<td>16KB/4-Way</td>
</tr>
<tr>
<td>L1 dCache</td>
<td>none</td>
<td>8KB/4-Way</td>
</tr>
<tr>
<td>Store Buffer</td>
<td>(8)</td>
<td>8</td>
</tr>
<tr>
<td>Forwarding</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Speculative Issue</td>
<td>No</td>
<td>Memory</td>
</tr>
</tbody>
</table>

issued the remote instruction may be stalled. The remote memory instruction travels across the SW1 or SW2 and this may cause contention.

2.2.6.5 Sample Parameters

The parameters used for the LWP cores in this work were chosen to reflect a reasonable, conservative design which emphasizes simplicity. It is similar to the PIMLite[20] processor in that it shares a close coupling the memory and a simple processor.

It is also comparable to the Niagara[109] processor in that it uses a simple multithreaded pipeline. Table 2.2 summarizes some processor parameters. The proposed LWP and the Niagara core use short pipelines and have hardware support for a limited number of thread states. They both contain instruction caches
which are relatively small when compared with conventional processors. The Niagara also contains a small L1 data cache. Both processors allow full forwarding in the pipeline. The Niagara processor allows 8 outstanding memory operations in its store buffer, the LWP allows one memory instruction to be outstanding per thread. One significant difference is that the Niagara processor allows limited speculative issue and execution for memory instructions. Though there are some minor differences in the specifics of the cores, their general similarity would indicate that the parameters chosen for the LWP are well within the current state of the art.

2.2.6.6 Power Consumption

Though a fully detailed power analysis of the proposed LWP architecture is beyond the scope of this work, it is possible to make some broad statements about power efficiency. This is most useful when comparing the LWP design to that of a conventional processor.

First, because the LWP is closer to memory it may be more efficient. Because it is on-chip, the number of remote memory accesses should be decreased, thus avoiding driving high-capacity off-chip busses. Secondly, avoiding caches (made from energy-expensive SRAM) will also improve power performance.

Secondly, the simplicity of the LWP microarchitecture is more efficient than a conventional processor. This is because the LWP discards several features found in more complex processors such as branch prediction, caches, and Out-of-Order-Execution (OOE) logic. This last point (OOE) is especially important as it means removing reservation stations, broadcast busses, and highly expensive multi-ported register files.
2.3 Software: The Incumbent

An analysis of scientific benchmarks such as the NAS Parallel Benchmarks [213] or ASCI Purple [157, 186] and many production workloads [154–156, 166] shows that scientific computing is dominated by FORTRAN [48] and C [101]. Older applications especially tend to be FORTRAN, while newer applications are more likely to use C or even C++. For parallelism, the Message Passing Interface (MPI) is commonly used.

2.3.1 Application Analysis

Analysis of how an application performs is probably as old as computers themselves.

2.3.1.1 Instruction Mix and Parallelism Limits

Some work has focused on the mix of instructions which an application executes [69]. This instruction mix analysis was instrumental in the development of the RISC architecture [83] as it revealed that compilers tended to use simple instructions.

As superscalar machines began to exploit instruction-level parallelism, some instruction mix studies evolved into to parallelism limit studies. These studies tried to find an “upper bound” on the parallelism extractable from a given instruction flow due to its data flow or control requirements. [184] examined the limited imposed by data flow and found that about two instructions could be issued per cycle, if a very wide (8+) processor was used. [111] found that control flow can have a very significant impact on performance, especially for non-numeric
codes. However, loop unrolling and speculative execution can alleviate much of this bottleneck.

2.3.1.2 Working Sets

Defining the “working set” of an application has been a common focus of research aimed at improving memory performance. This research into working set characterization has mirrored changes in general computer architecture research. Research on working sets in the 60s, 70s, and 80s, such as [46], were motivated by the need to develop efficient algorithms to swap virtual memory pages from main memory to a backing store. These works defined the “working set size” as the number of virtual memory pages touched over a given interval of instructions.

As efficient algorithms were developed and main memory got larger, the emphasis of later working set work changed, motivated by the growing difference between main memory and processor performance. Later work, such as [66] or [42], used working set analysis to illuminate memory hierarchy design or compiler optimizations [67, 78]. As a result, the definition of working set changed. [170] showed that applications tend to have a hierarchy of working set sizes, and defined them as sizes of memory which could contain a significant portion of the programs’ data.

2.3.1.3 Methodologies

Just as the focus of application characterization has changed to fit the focus of architecture research, the methodologies used to explore applications has changed. Earlier papers [46, 64, 181] developed analytical models to estimate program behavior. For example, these models were used to predict page reuse
and inform virtual memory paging. Measurements and validation were performed with performance counters or small traces [168].

Later papers relied more upon trace analysis [128], full system simulation, or hardware performance counters [100]. Analytical models were sometimes developed to avoid slower simulation [99]. Characterization was generally done in terms of cache performance, generally looking at cache sizes up to a megabyte [170].

2.3.1.4 Benchmarks

Though defining a “typical” application is always a challenging task, many benchmark suites have been developed and studied over the years. The SPEC suites have seen extensive working set characterization. [66] found SPEC92 benchmarks to have less than 5% cache miss rate with less than 40KB data cache. [78] used hardware counters to compare SPEC CPU2000 performance between different compilers. SPEC has also been used as a baseline when comparing against other workloads. [113, 128] used SPEC to compare various commercial, scientific, and desktop applications.

Database and OLTP processing have also been the subject of many characterization studies [11, 42, 100, 120]. Several other benchmarks have also been the target of characterization. [214] analyzed the SPLASH2 benchmark, finding that the first important working set is usually less than 16KB. [9] looked at the OMP2001 benchmarks under different scaling conditions.

2.3.1.5 NAS Parallel Benchmarks

This work utilized the NAS Parallel Benchmark Suite (NPB). The NAS Benchmarks are a suite of several scientific kernels and small applications. They are
written in C and FORTRAN. The suite is available in several versions: a serial version; a version parallelized with MPI; and a version parallelized with OpenMP.

Some of the NAS codes include:

**BT** is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations.

**SP** is a simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions.

**LU** is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block Lower and Upper triangular systems.

**FT** contains the computational kernel of a 3-D fast Fourier Transform (FFT)-based spectral method.

**MG** uses a V-cycle MultiGrid method to compute the solution of the 3-D scalar Poisson equation.

**CG** uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix.

These benchmarks can be configured to different sizes. For most of this work the 'W' or workstation-size data set is used.
2.3.1.6 Sandia “Benchmark” Applications

This work uses several applications as benchmarks for performance and communication studies. These applications are taken from common workloads at Sandia National Labs and are probably representative of numeric workloads at many high performance computing centers. The benchmark applications are:

- **ALEGRA**: ALEGRA[26] stands for Arbitrary Lagrangian Eulerian / General Research Applications. It is a multiphysics simulation code based on the Nevada finite element code framework.

- **CTH**: CTH is a multi-material, large deformation shock physics code used extensively to study armor/anti-armor interaction, warhead design, and weapons safety issues. It consists of roughly 500,000 lines of code, mainly in FORTRAN. It can operate with or without adaptive mesh refinement, and traces of both modes were collected. cth-2gas and cth-efp operate without AMR. cth-2gas does use AMR.

- **Cube3**: Cube3 is a benchmark built on the Trilinos[85] solver framework. The benchmark creates a beam, assembles a linear system, and performs a solve. Trilinos is an object oriented framework, written in C++. Two inputs are used. cube3-crs solves a 55x55 sparse compressed row system and cube3-vbr which solves a 32x16 variable block row system.

- **LAMMPS**: LAMMPS[153] is a parallel molecular dynamics code aimed at atomic or molecular simulations. The 2001 version used here consists of about 30,000 lines of Fortran. The lmp-chain input deck simulates a system of bead-spring polymer chains consisting of 810 atoms and lmp-lj solves a simple Lenard-Jones system.
• **sPPM**: sPPM is a benchmark from the ASCI Purple benchmark[157] and as the 7× application list for ASCI Red Storm. It solves a 3D gas dynamics problem on a uniform Cartesian mesh using a simplified version of the PPM (Piecewise Parabolic Method) code.

• **zChaff**: zChaff uses the Chaff heuristic [135] to find solutions to a boolean satisfiability problem. The input used here consists of 1532 boolean variables and 132295 clauses.

### 2.3.2 FORTRAN

FORTRAN, in its various incarnations, is still the powerhouse of scientific computing. Designed for easy parsing and to allow the compiler to concentrate on “more interesting” aspects of compilation, earlier versions lack many features of more modern languages such as dynamic memory access, recursion, or objects. However, the lack of these features did not seem to bother the majority of scientific and engineering programmers. Later extensions and revisions to FORTRAN have added many of these features, but care has been taken to not tradeoff performance when adding features.

### 2.3.3 MPI

The Message Passing Interface (MPI) [129] is the standard message passing programming interface for high performance parallel computing. Because it is network independent, implementations of MPI are available for many networks and shared memory machines. Additionally, because it is an open standard, hardware vendors are able to craft optimized versions for their custom hardware.
MPI provides a communication facility, but leaves it up to the programmer to partition the program to run in parallel. Though it provides some collective communication patterns, it does not mandate any particular style or communication pattern. Because of its dominance in high performance computing, MPI is covered in greater detail in Section 2.4.

2.3.4 OpenMP

The OpenMP standard [13] is a popular API for programming parallel shared memory architectures. Many compilers from Intel [94, 95], IBM, and others support the OpenMP standard on a variety of platforms.

An OpenMP program looks similar to a serial program in C, C++, or FORTRAN. The only difference is the addition of directives\textsuperscript{2} which are used to indicate parallel sections of code. This parallel section contains code that can be executed on multiple processors. Parallel sections most often contain loops. If these loops have a finite and known number of iterations, they can be parallelized and executed by a number of threads. OpenMP provides many options for how the loop iterations are distributed between threads (see Section 6.3.3.1) and how program variables are treated. In a parallel section, program variables can be kept private to each thread, shared between threads, or used in more a complex manner (such as being used in a reduction).

Because the serial code is retained, OpenMP parallelism can be added incrementally to an existing code. Also, the same codebase can be reduced to a serial version simply by telling the compiler to ignore the OpenMP directives.

\textsuperscript{2}also called \#pragmas in C/C++
Though originally targeted at homogeneous SMP systems, using OpenMP in heterogeneous and CMP environments has been explored in a number of works. [60] studies the effect of asymmetry in multicore processors on the performance of commercial workloads using SPEComp. [58] evaluates the performance of OpenMP applications for Simultaneous Multi-Threaded (SMTs) processors and multicore processors. The paper recommends more flexible OpenMP scheduling options to take advantage of different processor characteristics. [118] proposes a series of extensions to OpenMP directives to account for heterogeneity to demonstrate speed up on System-On-a-Chip systems. Several research efforts have also ported and evaluated OpenMP on a number of heterogeneous/multicore processors, including BlueGene/Cyclops architecture [53, 93] and the IBM CELL processor [38].

OpenMP has several advantages. It provides good performance and scales well on smaller systems. It can be adopted incrementally, and does not seriously complicate the code (unlike MPI). OpenMP is portable to a range of systems. It can be used in combination with other parallel programming models, such as a mixed OpenMP/MPI system. Also, it is widely seen as a convenient programming model.

However, OpenMP is still a minority player when compared to MPI. Of the top 20 applications run on NERSC machines, only 4 use OpenMP. Surveys of 116 users at SDSC found only 15% use OpenMP. A similar study at NCSA of 300 users found 14% use OpenMP[74]. Overall, OpenMP is seen as convenient, but not scalable to the hundreds or thousands of nodes high end scientific computing requires. Mixed models, using both OpenMP and MPI, evidence good performance, but many users don’t like having to manage two levels of parallelism. Additionally, the dominance of MPI creates a “vicious cycle”[121]. Because MPI
is dominant, programs are written for MPI, and supercomputers are procured based on their MPI performance. Because programs are written for MPI and systems are optimized for MPI, MPI continues to be dominant.

2.3.5 C/C++

Originally targeted for system development, C (and C++) are growing in popularity in scientific computing. This probably reflects a generational change as much as any inherent advantage of the languages. Like FORTRAN, C and C++ are sequential control-flow languages. C++ is generally a superset of C which adds support for object-oriented and generic programming.

One notable feature of the C++ standard is the Standard Template Library (STL)[180]. The STL is a templatized library of common data structures and algorithms. This library relieves the programmer from having to implement linked lists, associative arrays, sorting algorithms, and other commonly used components. This allows many concepts of Generic Programming to be supported.

2.3.6 HPCS Languages

The DARPA High Productivity Computing Systems (HPCS)[44] program, started in 2001, is aimed at stimulating a generation of high performance computers in the 2010 timeframe. One interesting focus of the program is on improving “productivity” rather than simply improving performance[74]. Part of this focus entails an effort to reduce the cost and time of developing new high performance applications. In addition to a number of innovative hardware architectures, it was required that new languages and programming models be examined as well. The
three phase two competitors, Cray Inc, IBM, and Sun are respectively developing
the Chapel[45], X10[174], and Fortress[123, 190] languages to meet this criteria.
All the the HPCS languages share a core of similar traits:

• Multithreaded programming is a basic concept. They all use some variant
  of a Partitioned Global Address Space (PGAS) model, similar to UPC or
  Split-C.

• Locality awareness is added to the basic shared memory outlook. Data and
  threads are given some notion of location. This allows the data layout to be
  specified to some extent.

• Object Oriented Programming is standard. “Objects” are supported in all
  of the languages, though are not always required.

Additionally, the languages share other criteria, such as garbage collection and
various levels of support for generic programming. They also provide a variety of
syntactic and language features to make it easier to change the number of nodes
the program runs on, the data distributions, and array types without extensive
rewrites of the code.

Each of the language development teams has noted two common problems.
First, the difficulty in designing a language which can run efficiently on a range
of systems, from uniprocessor workstations to commodity clusters to specialized
hardware has proven difficult. Sun’s Fortress and IBM’s X10 try to overcome this
by using a virtual machine to continually optimize the program’s execution and
adapt it to its current environment. Another problem is related to the performance
of modern networks. The small data transfers generated by a simple global address
space view tend to perform poorly on modern networks. A solution to this is to
aggregate small messages into larger ones. The languages all attempt to do this by a combination of compiler, language, and run-time optimizations.

2.3.7 Domain Specific Languages

Though high performance computing tends to be dominated by C/C++ and FORTRAN, a variety of domain specific languages and tools are used for scientific exploration. Often, these tools are the first choice for addressing a problem, and are used for prototyping a solution which is later converted to C/C++/FORTRAN for performance reasons.

- **Spreadsheets**: Originally designed for business and accounting, commercial spreadsheets are used for a variety of science and engineering tasks. Their integrated graphing and visualization packages are often much more accessible than similar stand alone packages.

- **MATLAB**: MATLAB has found use in almost all fields where matrix mathematics is performed. A variety of packages for specific mathematical functions are available either commercially or freely. MATLAB also comes with a simple GUI construction library.

- **perl**: Originally aimed at system administration and text processing tasks, the perl language has found some use by geneticists for DNA sequence matching.

2.3.8 Software tools

A variety of tools are available to help in the debugging and optimization of software.
• **Debuggers**: The most common tool for debugging a program is the debugger. Common features include the ability to stop a program and observe or change its state[189]. Well developed on serial systems, parallel debuggers are still an evolving field. Issues such as nondeterminism make parallel debugging notoriously difficult. As computers continue to scale to hundreds and thousands of processing nodes, devising an effective user interface may become a major concern.

• **Runtime Optimization Support**: A small set of tools for dynamically optimizing program execution at run time exists. Most of these tools take the form of Just-In-Time compilers for the Java language, however there are other systems, such as HP’s Dynamo[10], which are language independent. These tools can provide useful optimization by noting that tend to manifest only at runtime, such as branch tendencies.

• **Profilers**: Simple profilers, such as gprof[73] are often the programmer’s first line of optimization. Most act similar to a debugger, automatically stopping a program at regular intervals and analyzing its call stack. This can provide an accurate view of where the program is spending its time, and provide a good idea of where bottlenecks are occurring.

• **Hardware Performance Monitors**: A more advanced variant of the profiler, hardware performance monitors, such as PAPI or CHUD, use performance counting registers in the processor or memory controller to note certain events [25]. These events can then be correlated to the sections of code which produce them[6]. In this manner the programmer can gain spe-
specific information such as which section of code is causing the most branch mispredictions.

2.3.9 Software Design Patterns

Programming languages present their users with a set of primitive syntax structures - loops, functions, arrays, etc... - which define the language. These primitives can be combined in an infinite number of ways to produce useful programs.

However, in practice programmers tend to construct programs using familiar paradigms and practices used by others. An analogy put forth in [177],

For instance, automobile designers don’t design cars using the laws of physics. Instead, they reuse standard designs with successful track records. The extra few percent of performance available by starting from scratch typically isn’t worth the cost.

Many programming patterns refer to algorithms [37], common data structures [211], or network programming [161, 194]. They may be formally defined[87] as patterns, or merely be commonly accepted programming practice. These patterns are generally language independent, though their implementation may be easier in some languages than other.

Patterns are often defined with certain characteristics:

- **Problem**: The underlying cause of difficulty. (i.e. Memory is slow)

- **Context**: The specific situation in which the problem is encountered (i.e. Working with a large data set)

- **Forces**: Additional constraints upon a solution. (i.e. Faster memory is too expensive)
• **Solution**: How to circumvent or avoid the problem. (i.e. Use a cache or prefetch data)

• **Resulting Context**: Tradeoffs and concerns which present themselves after the solution is used. (e.g. Some access patterns thrash the cache, cache coherency protocols are complex)

These patterns aid productivity by allowing programmers to avoid having to reinvent the wheel (or, more precisely, the linked list) for each new program. Additionally, design patterns aid maintainability by often trading performance for maintainability. A commonly used pattern will be easier for a future maintainer to recognize than a novel or obscure mechanism.

The existence of these patterns creates an opportunity for compilers. Though it is necessary to accept all legal code for a given language, it is a reasonable assumption that certain patterns will occur frequently in programs. Building compilers (or novel architectures) to recognize and optimize for these cases can improve performance.

2.4 MPI

The Message Passing Interface (MPI)[129] standard is the dominant mechanism for parallelizing high performance applications[74]. MPI uses a message passing model where data is explicitly transferred from node to node. This model maps efficiently to modern networks which prefer large transfers of data to small messages.

A number of implementations of MPI exist. There are several open source implementations including LAM-MPI [29, 188] and MPICH [75, 76]. Also many network or cluster vendors produce their own proprietary or optimized versions[18].
A major advantage of MPI is that it is source-portable across implementations. This allows it to be used on a wide variety of platforms and with a wide variety of networks ranging from simple ethernet connections to fast shared memory machines.

2.4.1 Syntax, Semantics, & Implementation

MPI functionality is accessed through function calls to a library. Official Language bindings exist for C, C++, and FORTRAN, as well as unofficial bindings for other languages. Execution of an MPI program generally begins by starting a process on each node of a multiprocessor system. Each process has its own unique integer identifier assigned to it. This identifier is called a rank and can be used to specify the source and destination of messages. Additionally, ordered groups of processes can be constructed. These groups are commonly called communicators. Each process in a communicator is assigned a rank specific to that communicator. The program starts with one communicator, MPI_COMM_WORLD, which contains all the processes. Additional communicators can be defined which contain a subset of MPI_COMM_WORLD.

2.4.1.1 Simple Message Example

When sending a message, operations must be performed on both the sending and receiving side. For simple point-to-point messages, the sender calls MPI_Send. Six arguments are specified:

---

3. The MPI process is usually the same as a UNIX process with its own address space and OS state. However, the MPI standard does not mandate this and it is possible that an MPI process be a thread within a process or some other construct entirely.

4. Technically, it is only correct to talk about a rank in terms of a specific communicator. However, the term 'rank' is often used interchangeably with 'process' — i.e. by default, 'rank' refers to the rank in MPI_COMM_WORLD.
1. The local buffer containing the message to be sent (indicated by a pointer to memory),

2. The number of items in the message,

3. The type of the items (this can be a standard type such as 'integer' or 'double precision floating point' or a user-defined type),

4. The message destination (indicated by a rank),

5. An arbitrary 'tag' value determined by the user,

6. The communicator to which the receiver rank belongs.

There are also variants of sending for non-blocking sends and for different buffering options.

For receiving a message, two steps must be taken. First, a message request must be posted by the rank expecting the message. This informs the MPI library that the rank expects to receive a message. For simple point-to-point communication, this is done with the MPI_Irecv() function. Arguments are:

1. The local buffer to place the incoming message,

2. The number of items in the expected message,

3. The type of the items,

4. The source rank of the incoming message (this can also be a wildcard value which matches a message from any rank),

5. The tag of the incoming message (this can also be a wildcard value),

6. The communicator to which the sending rank belongs.
After MPI_Irecv() posts the “request”, the MPI_Test() or MPI_Wait() function can be used to poll or block on the actual message arrival.

In addition to point-to-point communication, MPI also includes a number of collective communication operations. These are performed on all the processes within a communicator. They include operations for reductions, barriers, broadcasts, and a variety of other data exchange operations.

2.4.1.2 Message Reception Internals

Internally, receiving a message requires several steps. Often, a node may have several messages in various stages of processing at once. This can lead to complicated context switching and “juggling” to perform the necessary accounting. This accounting revolves around three key data structures — the Posted Receive Queue, the Active Receive Queue, and the Unexpected Message Queue. These structures contain message requests\(^5\) and other state related to the message state (See Table 2.3). These queues are generally implemented as linked lists for reasons detailed in Section 4.1. The message reception usually follows a simple process:

1. MPI_Irecv() gives the MPI a message request. The MPI checks the Unexpected Message Queue to see if a matching request has already arrived. If not, the new request is added to the Posted Receive Queue.

2. The MPI periodically checks the network to see if any new messages have arrived. If so, it reads the message’s header information to determine where the message came from and its tag.

\(^{5}\) also called message envelopes
<table>
<thead>
<tr>
<th>Queue Name</th>
<th>Contents</th>
<th>Searched When</th>
</tr>
</thead>
<tbody>
<tr>
<td>Posted Receive</td>
<td>Message requests which have been posted and are waiting for their message to arrive</td>
<td>Message Reception</td>
</tr>
<tr>
<td>Active Receive</td>
<td>Messages which have been matched and are being received</td>
<td></td>
</tr>
<tr>
<td>Unexpected Message</td>
<td>Messages which have arrived but did not match anything in the Posted Queue</td>
<td>Message Posting</td>
</tr>
</tbody>
</table>

3. Given the message’s source and tag, the MPI traverses the Posted Receive Queue, and tries to find a matching posted request. If no match is found, information about the incoming message is placed on the Unexpected Message Queue. If a match is found, the matching request is removed from the Posted Receive Queue, and placed on the Active Receive Queue for further processing.

4. The MPI processes each item on the Active Receive Queue and tries to advance it towards completion. For a newly arrived message, this usually entails starting a DMA transfer or copying additional data from the network and into the buffer specified to `MPI_Irecv()`.

5. Once the data transfer (DMA or otherwise) is complete, the message can be marked as completed. Information on the message’s exact size, source, and tag is copied to a structure so that the application can retrieve it later.
6. Once the message is complete, the user’s call to `MPI_Wait()` or `MPI_Test()` will indicate that the message has been received.

2.4.2 Optimizations

MPI allows a number of performance optimizations. These range from internal protocols for handling messages to offloading much of the MPI library to an auxiliary processor to changing the application’s communication patterns.

One common optimization deal with message length. Short messages (below a few tens of kilobytes) are usually sent “eagerly” — in one message block. However, for longer messages it is often necessary to make sure that the receiving node has the resources available to handle the message. This usually means a buffer capable of handling the incoming data. For this case a “rendezvous” protocol is usually employed. With rendezvous, only the message header is initially sent. If the receiver has sufficient room for the message it will send back an “OK to Send” message. Only then will the sending node transfer the message data.

MPI collective operations\(^6\) have seen considerable optimization effort. Many of these optimizations are in the communication pattern and try to reduce the number of communications required or try to remove bottlenecks. For example, turning a broadcast from a simple one-to-many into a tree distribution may reduce the number of messages. Also, collectives may try to take advantage of special network topologies. For example, a network of SMPs may try to complete the “cheap” processor-to-processor communications before going over the network[102].

\(^6\)Communication operations which involve more than two MPI processes, as opposed to point-to-point.
Another common technique, especially for high performance networks, is to offload as much MPI processing from the CPU onto the network interface (NIC). This is done in Quadrics [151] and Myrinet [14]. This has been used to accelerate MPI protocol processing [17, 122, 203] and collective operations [27, 133]. A related approach adds special hardware to the NIC to accelerate MPI processing [206].

On the application level, the technique of pipelining messages into multiple segments to overlap computation and communication was explored in [43]. It noted significant promise for increasing performance, but noted the difficulty of finding the correct parameters for how to break apart messages. Also, it noted the negative effects on code size and maintainability.

2.5 Software: A Post-Mortem of Parallel Technology

A number of software technologies and alternative programming models have attempted to shape how parallel programs are constructed. Though many of them are successful research languages, they have not made huge in roads in mainstream scientific computing.

Though these technologies have taken diverse paths, they all address similar issues of handling memory latency, directing synchronization, performing load balancing, and expressing parallelism.

2.5.1 Split-C

Split-C [39] attempts to address the problem of expressing parallelism by extending the syntax of the C programming language. Aimed at SPMD programs on distributed memory multiprocessors, Split-C relieves the programmer from the
task of specifying communication patterns and data addressing. Split-C divides
the address space into two components - a global address space which is dis-
tributed across the system and a local address space where a program’s stack and
local heap is allocated.

To facilitate parallel programming, Split-C provides several features:

- **Control Model**: Split-C uses a SPMD (single program, multiple data)
  model. Each processor executes a single process. Processes can communicate
  through the global address space, and through synchronization primitives.

- **Split-phase Assignment**: Similar to a split-phase memory transaction,
  split-phase assignments allow a programmatic way to overlap communication
  and computation, similar to futures[198]. For example, the statement
  \[
  a := *b;
  \]
  would begin to load the variable \( a \) with the data pointed to by the global
  pointer \( b \). However, this operation could not be guaranteed to complete
  until a synchronization operation is called, after which \( b \)'s value is made
  available. In the mean time, other computation can occur.

- **Signaling Store**: Stores a value into the global address space and signals
  the owner\(^7\) of that data address that the store has occurred.

- **Bulk Transfer**: Large data structures can be easily “pulled down” from
  the global space to a local space for a program to operate on.

- **Spread Arrays**: Arrays can be specified so that they will be distributed
  amongst different processors in the system. Programmers can then trans-
  parently access the array without worrying about the exact communication

\(^7\)Where “owner” is the process to which the data is local
mechanisms. They can also define operations which will only operate on the 'local' part of the array.

2.5.2 UPC & Co-Array FORTRAN

Two languages which are similar to Split-C are UPC and Co-Array FORTRAN. Both of these languages use the SPMD model and allow data to be declared as private and local or as distributed and globally accessible. They both provide synchronization primitives to control the program’s operation.

UPC introduces the *shared* keyword to standard ANSI C to indicate memory which is distributed between processor nodes. The block size at which memory is distributed can be specified, and control structures, such as `upc forall` exist to allow a process to iterate over a section of the global memory to which it has affinity[33]. UPC allows synchronization through barrier operations, locks (similar to semaphores), and different memory consistency models[110]. In the “relaxed” memory consistency model synchronization must be performed explicitly by the user. The “strict” model implicitly adds synchronization routines whenever global data is accessed. UPC also has support for dynamically allocated shared memory structures.

UPC has added increased support for common collective operations[212]. Two classes of collective operations are specified in the specification: relocalization and computation. Relocalization operations include data broadcasts, scatters, and permutation functions. Computational operations include sorting and parallel prefix functions.

Co-array FORTRAN[144] extends the FORTRAN 95 language into a parallel SPMD language. Like UPC and Split-C, it allows data to be specified as local
and private or global and shared. Synchronization is accomplished through barrier operations and critical sections. Additionally, extensions are added to allow efficient parallel I/O. Co-Array FORTRAN has been used on a number of linear algebra and scientific tasks[32].

2.5.3 SISAL

Instead of expanding existing control-flow languages, the SISAL[131] project took a more fundamental approach. SISAL (Streams and Iteration in a Single Assignment Language) is a functional parallel language which allows parallelism to be expressed more naturally by removing side effects from programs. SISAL showed significant possibility in allowing programmers to express parallelism easily and without much knowledge of the underlying hardware[63].

However, SISAL suffered from some problems. First, because it was a functional language it presented a much steeper learning curve to programmers of conventional languages such as C or FORTRAN. Secondly, because of its functional heritage it made dealing with I/O difficult. Because I/O is essentially “one big side effect” it is hard to fit into a pure functional model. Lastly, SISAL did not really address issues of memory locality, thus limiting its use on distributed memory machines.

2.5.4 Parallelizing Compilers

Vectorizing and parallelizing compilers attempt to transform loops to use vector operations [147] or to distribute independent iterations of a loop between several parallel threads [199].
Though compilers can parallelize a number of loop types, there are some conditions which may prevent parallelization or vectorization:

- **Data Dependence**: Perhaps the most common impediment to parallelization, if iterations of a loop are dependent on the results of previous iterations, the loop generally cannot be parallelized.

- **Function Calls**: A special case of data dependence, loops containing function calls often cannot be parallelized unless the compiler can determine there are no side effects from the function call. This is often the case when using compiling modules separately.

- **Array Aliasing**: Another special case of data dependence. Many languages, such as C, lack a mechanism to indicate if two address pointers are pointing to the same array in memory. Unless this information can be determined, the compiler cannot be certain that iterations are independent.

- **Can’t Determine Loop Iteration Count**: If the number of iterations of a loop cannot be statically determined at compile time, some compilers cannot parallelize the loop. This also applies to loops which may terminate prematurely (ex: searching an array).

- **Loops Too Small**: Depending on the threading mechanism, a loop with too few iterations may not be worth parallelizing due to the overhead of starting new threads.

2.5.5 Dataflow Languages: TAM

One outgrowth of the hybrid dataflow machines, is the Threaded Abstract Machine (TAM)[40]. TAM uses a data-flow/control-flow hybrid model to enable
dataflow execution without hardware support. TAM emerged from the realization that economic and technological constraints limited the feasibility of new dataflow implementations in hardware. Instead, TAM attempts to use existing conventional hardware and layer a dataflow model on top of it.

TAM has been ported to large parallel machines, such as the CM-5. Activation frames are stored in the local memory of a processor, and heap data is spread amongst processors.

Though TAM is able to successfully scale some applications, it suffers from high overheads imposed by the underlying hardware. Often, upwards of 40% of TAM instructions deal with thread scheduling and control, which significantly reduces the benefits of parallelism.
CHAPTER 3

SIMULATION ENVIRONMENT

The primary experimental methodology used to investigate the techniques explored in this thesis is instruction-level simulation. A execution-based simulator (the Structural Simulation Toolkit) is constructed from existing and new models to encompass the configurations required for the above experiments.

The level of simulation was chosen to ensure a greater attention to realistic modeling than analytical modeling would allow, while also permitting greater flexibility and productivity than full VHDL or hardware simulation would allow.

The Structural Simulation Toolkit (SST) is an architectural simulator implemented in about 45,000 lines of C++. It is composed of four primary elements (see Figure 3.1): the Front End, which emulates the execution of a program; the Back End, which models architectural components of the system; the Processor/Thread Interface, which allows the front and back ends to interact; and Enkidu[164], a component-based discrete event and synchronous simulation framework that coordinates communication between back end components and models the passage of time. To provide modularity and reconfigurability, it is possible to select a front end and choose a variety of back end components. This selection occurs when the simulation begins. This allows the user to explore a variety of hardware configurations while using the execution model best suited to the available toolset.
Put another way, the Front End emulates a program’s execution from a “software only” perspective. The Back End models the hardware components of the system, which provides timing and performance data. The bridge between these two is the Processor/Thread Interface.

3.1 Motivations

The Structural Simulation Toolkit was constructed after looking at a range of existing architectural simulators. Its three major goals are scalability, flexibility, and modularity. Scalability allows the simulator to examine parallel (both multithreaded and multiprocessor) systems as they scale. This examination is made more efficient by allowing hardware configurations and software features to be varied independently — flexibility. The simulator’s complexity is kept manageable by its modularity.
3.1.1 Existing Simulators

Many architectural simulators have been written to explore design issues on the processor or system level. These simulators represent programs by execution-based, trace-based, or stochastic mechanisms. They vary in level of detail, configurability, and focus. A major goal of the Structural Simulation Toolkit is to build upon the success of these previous works.

SimpleScalar\cite{SimpleScalar} is one of the most commonly used architectural simulation toolkits. It includes a number of execution-based simulators, ranging from simple execution to cache simulation to full simulation of an out-of-order processor and memory hierarchy (\texttt{sim-outorder}). A number of processor parameters can be adjusted, such as issue width, functional units, internal queue lengths, and cache characteristics. Though SimpleScalar only models a single conventional processor, several derivatives have extended its functionality. The SIMulator for Multithreaded Computer Architecture \cite{SIMCA} (SIMCA) was developed to explore multi-threaded architectures by augmenting the PISA\footnote{PISA is an ISA used in the SimpleScalar toolset. It is based on the MIPS ISA.} ISA to include instructions for thread creation and control. Another derivative, MP\_simplesim\cite{MPsimplesim} presented multiprocessor cache simulation capabilities. However, this simulator only modeled cache interactions and did not attempt to model detailed timing. Other multiprocessor simulators include Solo\cite{Solo}, RSIM\cite{RSIM}, ML-RSIM \cite{ML-RSIM}, Tango\cite{Tango}, and MINT\cite{MINT}.

Other simulators have been developed to enable specific functionality. For example, SimOS\cite{SimOS}, Talisman\cite{Talisman}, and ML-RSIM\cite{ML-RSIM} are capable of supporting the execution of an OS. SPIM\cite{SPIM} was developed as a teaching tool for exploring the MIPS ISA. Others, such as \texttt{simg4}\cite{simg4} and \texttt{simg5}, were developed to model a
specific processor (the PowerPC 7400 and 970) in detail. Both of these simulators are trace-based, and provide a high level of detail, but lack a high degree of parameterization. Additionally, they do not attempt to simulate a full system with multiple processors or network.

The ASIM[49] performance model framework is comprised of a set of modules which can be composed to form different architectures. A novel feature of ASIM is the partial separation of the performance model for system components from the program execution. This allows the hardware parameters and software to be changed without requiring extensive redesign.

Other modular simulation efforts include the Liberty Simulation Environment [207], which has developed a number of modules in its own LSS language, and Microlib which provides a number of modules in SystemC[145]. Both of these efforts are focused on low level simulation of processor and memory components at the RTL-level of detail.

The message PAssing computeR SIMulator, PARSIM[197] was developed to explore algorithms and network topologies for parallel computers. It models program execution as a generalized algorithm divided into computation and communication. Processor speed and network characteristics can be parameterized, but no attempt is made to model the internals of the processor.

These existing simulators provided a range of options, levels of detail, performance, and focus. For the experiments detailed here none of them covered all the key issues and so it was decided to construct a new simulator infrastructure, drawing from previous experiences.
3.1.2 Issues Related to Scale

The first goal of the Structural Simulation Toolkit was to look at issues related to scale. Many existing simulators were not scalable beyond a single processor. Their primary purpose was to explore issues related to a single core — such as cache performance, pipelining, instruction scheduling, and branch prediction. Also, most processor simulators focused on single-threaded designs.

The Structural Simulation Toolkit is designed to allow scalability on multiple fronts. First, it allows multiple threads to exist in the same simulation. Secondly, processing can occur simultaneously (in simulated time) in several different components. Thirdly, this parallelism can be explored at multiple levels (such as multiple functional units, multiple cores on a chip, multiple chips in a system, and multiple systems). Lastly, to realistically model this processing, heterogeneous systems must be allows. For example, a system containing different processor types for the CPU and network processor could be simulated. By including parallelism at multiple levels of the simulation infrastructure, it is possible to scale the simulation without rewriting the basic models. This differs from other simulators which are confined to a single thread or single processor, and which cannot cleanly simulate system-level parallelism.

3.1.3 Variable Hardware and Software

Another key goal was to allow new hardware and software features to be explored rapidly. This required that hardware models could change without requiring a completely new set of software tools, and software techniques could be tried without scrapping existing hardware models.
This was accomplished by keeping the hardware timing models and software emulation sections of the simulator independent, and defining a strict interface between them. This interface limits how much either of these sections “knows” about the other and allows them to keep their internal functioning hidden. For example, making the processor definition independent of the ISA allows different processor characteristics to be simulated without having to rewrite compilers. Similarly, making the definition for “thread” generic allows the processor to simulate instructions from a PowerPC trace, a PISA program execution, or a PowerPC program using the same codebase.

3.1.4 Modular Design

The complexity of the hardware models and software interpreters required a modular design to remain manageable. Hardware models are broken into a series of component objects. Software is represented by thread and instruction objects. This modularity allows significant code reuse. For example, the conventional processor used as a CPU (in Chapters 4, 5, and 6) is also adapted to be used as the NIC controller (in Chapters 4 and 5).

3.2 Model of Computation

An important characteristic of a simulator is the underlying model of computation [114] which defines how time is advanced and how components interact. Most architectural simulators use one of the following approaches:

- **Synchronous**, or time-stepped, models discretize time into fixed increments. At each increment, all components are evaluated and the effects
are propagated. Simulators using this approach include SimpleScalar [28], SPIM [112], or PARSIM [197].

- **Discrete Event**, or event-driven, approaches only evaluate transition functions if component inputs change, or if an event is received from another component in the system. This model is used in VHDL simulators.

- **Process-model** uses processes to model components. Processes represent the activities of some component, how it interacts with other components and how it advances its own state. The framework provides a facility to schedule execution of processes and switch between them to simulate simultaneous action [178]. Examples of this include CSIM and SSF.

- **Hybrid** simulators combine two or more simulation techniques. For example, Enkidu uses both time-stepped and discrete event models and SSF [119] uses both process-based and discrete models.

Discrete event simulation is popular for general simulation, battle space simulation, gate-level microarchitectures, and network simulation, where many components generate events at varied rates. Synchronous simulation is more popular in architectural processor simulation, where every microarchitectural component would often generate an event every clock cycle.

When inter-component interaction is infrequent, discrete event simulations have an advantage, in that they can “skip” intermediate time steps between event arrivals. Note that in a pure discrete event simulation, if a component needs to change its state it must send an event to itself. However, when events are relatively infrequent, asynchronous simulation tends to incur less overhead.
Process-oriented simulation allows components to update their state without an explicit event. As such, they can be an elegant way of expressing frequent, repetitive, behavior (like a time-stepped system) while still allowing non-uniform time steps (like an asynchronous system). However, process context switching can become a significant source of overhead [119, 178].

Many works have looked at simulation speed for given simulations, or at the overhead provided by a particular framework [158]. Much of this work has focused on parallel discrete event simulation[65, 108], often focusing on the benefits of a particular language or framework such as SSF[119] or PARSEC [197]. A commonly used optimization in parallel simulation is to use a system of timestamps and rollback mechanisms to perform speculative simulation[117].

3.2.1 Hybrid Simulation

Modern processors can be represented as a series of buffers which store instructions and data, separated by logic which acts upon those instructions. Data flows from buffer to buffer according to a strict centralized clock. For architectural simulation, it is possible to say that all events take place in synchronization with this clock. Modern processors often have dozens of instructions in various stages of execution during each processor clock cycle. As a result, several transition events can be expected to occur each cycle.

Parallel architectures will often have two distinct types of transitions. Transitions within a processor occur several times a cycle. Additionally, there will be infrequent inter-processor communication events. For this reason a hybrid simulation framework which combines synchronous time-stepping and discrete event-

A transition event is an event which changes the state of the simulated component
passing provides the most flexibility and lowest overhead for parallel architectural simulation.

The Structural Simulation Toolkit is built around Enkidu, a hybrid simulation framework. In Enkidu, component objects represent each physical component of the system. Each of these components is evaluated every clock cycle, allowing it to advance its internal state. In addition, components can communicate by passing event messages to each other in an asynchronous manner. Like SystemC, Enkidu uses the object-oriented features of C++ to divide up simulation into many different components. Optimizing for the common case in architectural simulation, Enkidu provides low-overhead synchronous time-stepping to handle the bulk of the functionality. For the infrequent communication between components, the discrete event mechanism is provided.

Figure 3.2 contains pseudocode for Enkidu. This pseudocode consists of a while loop (starting in line 1), each iteration of this loop is one cycle of simulated time. A counter (cycle) keeps track of how many cycles have been simulated.
This counter is incremented each cycle (line 2). In each cycle, each component has a function invoked upon it (advance(), line 4). This advance() function allows the component to advance its internal state. The loop which iterates through the components (line 3) is low overhead. After each component has advanced its internal state, inter-component events are handled (line 6-10). Each event is represented by a structure which has two key fields: arrivalTime, which indicates the cycle at which the event should be delivered; and destination which indicates the component it should be delivered to. Events which are “due” in the current cycle are delivered to their destination component with the handleEvent() function (line 8). This function allows the component to perform whatever processing is required. After all events for that cycle are delivered, the loop begins again.

3.3 Modular Framework

The four elements of the Structural Simulation Toolkit are the Front End (Section 3.3.1), the Back End (Section 3.3.2), Processor/Thread Interface between (Section 3.3.3), and Enkidu (Section 3.2.1). The Processor/Thread Interface provides a series of object class definitions which allows the front end and back end to communicate the specifics of a program’s execution.

3.3.1 Front End

The front end is responsible for generating events representing instructions and threads by executing a program or reading a trace of a program’s execution. Information about these instructions and threads is requested by the back end so it may model timing. The front end also defines how the execution of these
instructions changes the programmer visible state. Put another way, the front end emulates the “software” or programmer visible side of the system.

The front end provides a loader which loads the program into the simulated memory before the simulation begins. Once simulation begins, the front end provides instruction and thread objects to the back end (Figure 3.3) via the interface described in Section 3.3.3. Most importantly, it determines how the program state (usually memory and registers) are modified by the instruction’s execution. This usually involves a lookup of the instruction in a “big case statement” to ascertain how state should be updated. This “big case statement” performs a similar function to SimpleScalar’s\cite{28} \texttt{ss.def} or \texttt{powerpc.def} files, or SPIM’s\cite{112} \texttt{run.c} — it maps instruction opcodes to behavior. Currently, three front ends exist. The user selects which to use when the program begins with a configuration file that is read at simulation startup.

- **PISA**: an execution-based front end which uses the PISA ISA and the ELF executable format. The PISA ISA is augmented with extra instructions to provide added functionality required to simulate experimental hardware. These instructions were added to explore low-cost thread spawning and different synchronization primitives, to allow a processor to communicate
with a subordinate processor on a NIC, to access specialized hardware such as network interfaces, DMA engines, or custom components, and to allow a processor to select between different address spaces. The PISA front end allows use of modified versions of the gcc or g++ compilers version 2.7.2.

- **PPC**: An execution-based front end which uses the PowerPC ISA[136] and the MachO[8] executable format. This PowerPC ISA is augmented in a manner similar to the PISA ISA. It also includes a small subset\(^3\) of the AltiVec vector extensions. The MachO format is the standard format for MacOS X executables and allows the use of binaries created by a number of modern compilers. It has been tested with gcc and g++ versions 3.1, 3.3 and 4.0.1, g77 version 3.4, IBM XLF 8.1 FORTRAN 77/90 compilers, the Omni FORTRAN compiler, and Absoft 8.0 FORTRAN 77/90 compilers.

  Additionally, a dynamic loader for the MachO executable format, was also created. This allowed the use of a wider range of modern compilers and was necessary to simulate large “real-world” scientific applications. This is because some support libraries may not be available in static form. This allowed the simulation of large applications such as CTH[50], a 500,000 line mixed FORTRAN/C/C++ shock physics application with adaptive mesh refinement and visualization capabilities.

- **Trace**: A trace-based front end which uses the tt6 PowerPC instruction trace format[7] generated by the amber trace generator[6]. A tt6 file contains a record of each instruction executed by the target program. This record contains the opcode, registers, immediate and any memory addresses accessed or the branch target. The Trace front end also supports limited

\[^3\]The subset includes the lvx, stvx, vspltw, cmpequh[,], and vand instructions.
speculative execution. Before the simulation begins, the trace is scanned and the address of each instruction, along with that instruction’s record, is recorded. If the back end mispredicts a branch and requests an instruction at the wrong address, the front end will return the instruction at the incorrect address, as long as that address is executed somewhere in the trace. Because most mispredicts still return a valid instruction address, the back end is able to simulate the effect of the speculatively executed instructions.

3.3.2 Back End

The back end is responsible for modeling the physical components (hardware) of the system. Specifically, it consumes instructions and threads generated by the front end and determines how long it would take for the specified hardware to execute them. The back end is composed of many different Enkidu components (see Section 3.2.1). These components represent physical components such as processors, networks, memory controllers, DRAM chips, or other system elements. Components can communicate through Enkidu’s (Section 3.2.1) discrete event message passing system.

Several back end components have been developed. Several of the key components used in this work are detailed in Section 3.4.

3.3.3 Interface

The Processor/Thread interface is the key bridge between the front end and back end. This interface defines three abstract object classes: processor, thread, and instruction. A processor is a back end component which can execute instructions belonging to one or more threads. Examples of processors include
the LWP Processor and the Conventional processor. Each front end defines a thread class and instruction class. Thus, each given processor object does not need to know if the instruction it is executing comes from a trace, a PowerPC executable, or a PISA executable.

The front end simulates how the program executes from a software perspective — it ignores hardware and timing details and just looks at how the state of its registers and memory is modified by the instructions. The back end processor simulates how the program executes from a hardware perspective — it ignores the specifics of what values are written where and focuses on modeling the timing details of which components are accessed, how long memory transactions take, and other microarchitectural details. The back end processor provides access to a storage area for the front end to store memory and register values. This allows multiple threads to interact on the same data.

3.3.3.1 processor / thread Interaction

The execution of an instruction – demonstrating the typical interaction between processor and thread – is shown in Figure 3.5. This execution may span
several Enkidu cycles. It occurs in a processor’s advance() function in Figure 3.2 (line 4). When simulating a parallel system, interactions like this may be occurring in multiple processors during the same simulated timestep.

In (a), the processor begins by requesting an instruction from the thread. This request can be for an instruction at a given address, or can simply be for “the next instruction to execute.”

The processor simulates the execution of the instruction in three stages: fetch, issue, and commit. These stages can be performed at different cycles, and they can be overlapped with other instructions. First, the processor calls the fetch() method on the instruction (b). The fetch() method may call upon the processor to access simulator state, usually to access the instruction memory. After fetch() has completed, the processor can query the instruction for information such as
the instruction’s type. This information can be used to model instruction timing and resource use.

A similar process is performed for the issue() stage(c) and commit()(d) stages. The processor calls a member function of the instruction. This function may invoke processor members to read or write state. After the instruction member function completes, the processor can query the instruction to find characteristics such as which functional unit(s) it requires, its register dependencies, and any exception.

Lastly, the processor directs the thread to retire the instruction(e) and perform any necessary cleanup. If the processor needs to rollback (see Section 3.3.3.2) the execution of one or more instructions (perhaps due to an exception or branch mispredict) it may do so with the thread class’ squash() function. This will rollback the thread’s state and perform cleanup.

3.3.3.2 Speculative Execution

To model many processors accurately, speculative execution must be simulated. To accommodate this, the SST uses a copy-on-write policy towards memory and register state. If a processor detects the start of speculative execution\(^4\), it informs the thread via the prepareSpec() function. The thread then treats future commit()s as speculative. Any state changes caused by these commit()s are written to a copy of the original state. When the speculation is over (i.e. the branch mispredict is detected) the thread and processor discard the speculative state copies. Because speculation is usually short, the size of the speculative state is generally quite small and manageable.

\(^4\)e.g. It mispredicts a branch.
### TABLE 3.1

**KEY COMPONENTS**

<table>
<thead>
<tr>
<th>Component Set</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional Processor</td>
<td>20000</td>
</tr>
<tr>
<td>Memory System</td>
<td>1400</td>
</tr>
<tr>
<td>NIC &amp; Network</td>
<td>2200</td>
</tr>
<tr>
<td>LWP</td>
<td>1900</td>
</tr>
<tr>
<td>Analysis</td>
<td>500</td>
</tr>
</tbody>
</table>

#### 3.4 Key Components

Several back end components have been developed. Many of these classes can be configured to interact and build upon each other (Figure 3.4). Four sets of components which are used extensively in this work are discussed below. Their code sizes are summarized in Table 3.4.

#### 3.4.1 Conventional Processor

A conventional, out-of-order, multi-issue processor based on SimpleScalar’s `sim-outorder`. It can use SimpleScalar’s memory model or can connect to a memory controller component (see Section 3.4.2). It can be configured to handle multiple threads with simple run-to-completion scheduling, or can run single threaded. The model can be parameterized to simulate different cache sizes, issue widths, and other execution parameters.
Modifying SimpleScalar’s sim-outorder involved three major changes. The first was decoupling sim-outorder’s instruction emulation mechanism from its timing model to accommodate the interface described in Section 3.3.3. Secondly, sim-outorder’s memory model was overly simplistic, only allowing a fixed latency to be used. This memory model was expanded to accommodate the more robust and detailed memory components described in Section 3.4.2. Thirdly, basic support for run-to-completion multithreading was added.

This model is used as a CPU core connected to a memory system in Chapters 4, 5, and 6. Chapter 7 uses an isolated core to establish baseline performance. Chapter 6 uses a variant which includes a simple snooping cache coherency policy to allow CMP operation.

3.4.2 Memory System Components

Memory system components were added to simulate the backing DRAMs for an LWP configuration and to connect “dumb” DRAM to conventional CPUs. The memory system contains two main components (Figure 3.6):

- A simple memory controller model which simulates bus contention, bandwidth constraints, communication latency, and DRAM address interleaving.

![Figure 3.6. Memory system components](image-url)
The memory controller connects processors and DRAMs. Incoming requests (from the processor) are queued as they arrive and serviced in a FIFO manner. Returns from the DRAMs are queued in a similar manner. The rate at which the queues are serviced allows the bandwidth through the controller to be managed. Additionally, the controller adds a fixed latency to incoming (or outgoing) request to simulate communication time. The memory controller also assigns memory requests to different DRAMs based on an address interleaving scheme.

- A model of a DRAM chip with a configurable number of DRAM banks. Open row\(^5\) latency and contention effects are modeled, and the number of open rows per bank can be adjusted. The number of DRAM macros per DRAM is a parameter. Each DRAM macro also contains one or more open rows. Two latency parameters are set – the latency for an open row “hit” and the additional latency for an open row “miss.”

3.4.3 NIC & Network Components

To simulate networked operation, a NIC and network model was devised. These components allow a system comprised of a Network Interface (NIC), CPU, and memory system to connect to other similar systems through a configurable network model. The network models simulate data, broken into packets, be transferred between nodes. The networks terminate in Network Interfaces (NICs) which model the processing of this data and its possible transfer to the memory of a simulated system.

\(^5\)Also called “open page.” A bank of DRAM usually consists of rows of 1024 or more bits. Often, when reading or writing the memory an entire row is read into a buffer before accessed. If several consecutive accesses occur to the same row, it can be left “open” in the buffer.
Figure 3.7. 2-D mesh router

The NIC is controlled by a conventional processor or LWP processor(s). It also models a separate NIC bus, connections to the network and main memory, on-board memory, and multiple Direct Memory Access (DMA) engines. The DMA engines perform block data movement between a NIC and main memory. Multiple Tx and Rx channels can be configured. The DMA engines are controlled by the NIC processor. This control can be through a polling or queue-based mechanism.

The overall model was loosely based on the NIC used in ASC Red Storm[30], but it is similar to many high performance NICs (such as Myrinet). The NIC model is used in Chapters 4 and 5.

Two network models were constructed. The first is a simple “latency injector” which connects a maximum of two components. When a data parcel (represented by an Enkidu event) arrives at the simple network from one node, it is resent to the other node after a fixed delay of $x$ cycles, where $x$ is a user supplied parameter.
The second network model is a more complex 2-D or 3-D mesh network modeled after the Red Storm network\textsuperscript{6}. This network models the individual routers and their internal queues. Data is represented at the flit level. Figure 3.7 shows a sample 2-D router from the more complex network model. The router is an \texttt{Enkidu} component which is connected to four other routers (\texttt{North}, \texttt{South}, \texttt{East}, and \texttt{West} in the Figure) and to a Host NIC. Each link has an incoming and outgoing buffer. As message flits arrive (again, represented by \texttt{Enkidevent}s), they are initially enqueued in the “incoming” buffer for that link. A routing core examines the incoming buffers in a round-robin fashion. If the buffer is not empty, it consults a routing table to determine where the contents should be transferred to. It then transfers the contents of the incoming buffer to the appropriate “outgoing” buffer. This transfer may take several cycles. Additionally, each cycle, outgoing buffers are checked. If there is any outgoing data in the buffer, it is transferred across the link to the next router, or to the host. A flow control mechanism using “fake” packets provides bandwidth constraints.

These components can be used together to simulate a modern distributed memory system (Figure 3.8). The NICs, containing a processor and DMA engine, connect CPU and main memory of a node to other nodes via the network routers. To transfer a message, one NIC starts a DMA from the node’s main memory to the network. The message travels through the appropriate routers to the destination NIC. The processor on the destination NIC will look at the message headers and direct its DMA engine to transfer the data to main memory. In the Structural Simulation Toolkit, all communication between components (e.g. main memory to NIC, or router to router) is performed with discrete events managed by \texttt{Enkidu}.

\textsuperscript{6}The mesh network model was developed by Elizabeth White, working at Sandia National Labs.
3.4.4 LWP Components

Light Weight Processing is modeled with two components. First, a Lightweight Processing Chip (LPC) contains multiple LWP cores and simulates the communication between those cores and off-chip. Second, the cores themselves model the execution of instructions.

The LPC model is a Processor-in-memory (PIM) chip that contains one or more LWP processors. The LPC also models instruction caches, on-chip DRAM or SRAM memory, inter- and intra-chip communication between PIM processors, and optional connection to a backing memory controller or a CPU. The size and associativity of the on-chip caches can be controlled, as can the ratio of LWP cores to instruction caches. Memory addresses is distributed between the different LWPs and on-chip memory by hashing. The size of these hash blocks can be varied.

The LWP component models a multithreaded in-order pipeline without branch prediction. It can be configured for different pipeline lengths, forwarding, and
thread scheduling mechanisms. It supports non-local memory accesses by remote instructions or thread migration. This component encapsulates the concepts for a proposed LWP set forth in Section 2.2.6.1.

The LWP components are used in Chapters 4, 5, and 6.

3.4.5 Analysis Components

In addition to components which model actual or theoretical hardware, it is also possible to construct components which do not model hardware, but simply consume instructions and perform analysis on them. This class of components is used in Chapter 7. In that chapter, a component consumes instructions, partitions them into small threads, and estimates their performance on a simple pipeline.

3.5 Performance

To measure the performance of the hybrid approach and the decoupling of the hardware and software, a simple program was benchmarked. This program was run under a variety of simulators and configurations of the SST. This program consists of a linked list creation and traversal (a memory intensive operation), a matrix-matrix multiply (a cache and floating-point unit intensive operation), and several system calls. These simulations were performed on a PowerPC G4. The purpose of this benchmark is to determine the speed at which the different simulators can simulate a given program.

Two metrics are used to judge performance. First is the number of host processor cycles required to complete a simulated cycle. That is, the number of processor cycles on the real-world G4 processor running the simulation compared to the number of cycles of the simulated processor. Simulators generally require several
Figure 3.9. Simulator comparison: the Structural Simulation Toolkit incurs only a small overhead against SimpleScalar
thousand “real-world” processor cycles to simulate a single simulated processor cycle. The second metric is the number of host (“real-world”) cycles required to simulate a single instruction.

Figure 3.9 compares the performance of the Structural Simulation Toolkit with both SimpleScalar and simg4, a trace-based simulator for the PowerPC G4 (7400). The Structural Simulation Toolkit trace-based front end uses the same format as the simg4 simulator, but the back ends are different. Three variants of the SST simulator are tested. SST-PISA which uses the PISA ISA, SST-PPC which uses the execution-based PowerPC ISA front end, and SST-Trace which uses the trace-based PowerPC front end. Above, the average number of host processor cycles required to complete a simulated cycle are shown.

Comparing the SST-PISA data point to the SimpleScalar data point indicates that the modularity of the framework adds approximately 10% overhead. This is a relatively low overhead for separating the instruction decode from the microarchitectural simulation. Using the PowerPC front end increases the overhead slightly per simulated cycle and the trace based front end is much heavier weight.

Moving to the lower half of Figure 3.9, gives an “instruction to instruction” ratio between simulated and host instructions. This effectively divides the data on the upper graph by the IPC achieved on the simulated platform. The performance relationship between the PISA based SimpleScalar and the PISA front end of the Structural Simulation Toolkit remains constant; however, the PPC front end and trace based front end show significant relative gains. This occurs because the PPC and trace based front ends use a newer compiler (gcc 3.3) rather than the older compiler (gcc 2.7.2) available for PISA.

7This is currently believed to be a difference in the level of optimization effort applied to the trace based front end.
CHAPTER 4

MPI and LWP

MPI, the Message Passing Interface (see Section 2.4), is the de facto standard for communication in high performance cluster computing. Considerable work goes into improving the communication patterns, software implementation, and network hardware which are brought together by the Interface.

Modern high performance network interfaces use programmable NIC processors to offload much of their MPI processing [4, 151]. This frees the CPU from what can be hundreds or thousands of instructions per message [165]. This offload is spurred by the evidence that messaging overhead has a large impact on application performance [41, 127].

Sadly, a large portion of the message overhead involves searching lists of MPI request records. To achieve high performance, these list traversals must be done quickly. These lists can grow quite long [16, 19] and the long traversal lengths are made worse because the NIC processors must have lower clock speed and less complexity than the CPU [205].

A solution to this is to utilize a better NIC processor. Such a processor needs to decrease the time required to search a request list. Additionally, it should be able to overlap multiple searches of the list to increase throughput. LWPs fit these requirements. An LWP has naturally low latency access to memory, so it traverses linked lists well. Memory accesses are wide and can be operated on
with a wide ALU, so multiple list entries can be searched rapidly with a handful of instructions. The LWP’s hardware support for multi-threaded and extremely fine-grained locking support enable simultaneous queue traversals to leverage the increase in bandwidth. Finally, LWP\textapos;s are less complex, so the area and the cost devoted to them will be minimal. In short, LWP\textapos;s are a natural fit to the needs of the embedded processor on the NIC.

This chapter explores how LWP can accelerate NIC-based MPI processing. A simulated NIC model (see Section 3.4.3) using a PIM processor is compared with a NIC using a conventional processor. An implementation of MPI was modified to leverage the LWP\textapos;s wide memory accesses, wide ALU operations, and multithreading capabilities. The results indicate a dramatic decrease in message latency in the presence of long queues and an substantial increase in the achievable throughput.

4.1 MPI Matching Procedure

The lists within MPI which must be searched are the posted receive queue and unexpected message queue (see Section 2.4). They are traversed under the following conditions:

- Whenever a receive request is posted, the MPI must check the unexpected queue to see if a message matching the posted request has already arrived.

- Whenever a message arrives from the network, the MPI must check the posted receive queue to see if the message is expected.

The MPI standard requires that a message only match a request if the message\textapos;s source, tag, and communicator match ([129] p21 lines 13-16). Additionally,
the user can specify “wildcard” values for the source and tag fields which match any value.

Furthermore, the standard mandates that message ordering be non-overtaking (p30 lines 44-48):

If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending.

These requirements hamper attempts to minimize the list traversal time by replacing the list (and its linear traversal time) with a tree or by hashing. Such attempts have been unsuccessful due to the added overhead, the possibility of “wildcards”, and the MPI ordering semantics. For example, using a simple tree structure would not be possible due to the ordering requirement.

4.2 LWP-Based NIC Design

The hardware setup for an LWP-based NIC replaces a conventional NIC processor with multithreaded, vector-capable LWPs. The modified MPI then makes use of these capabilities (threading and vector processing) to improve MPI performance.

4.2.1 Hardware Design

The baseline NIC (shown in Figure 4.1(a)) is representative of numerous modern network interfaces, including Quadrics [151], Myrinet [14], and the Red Storm system developed by Cray and Sandia [4, 30]. Such NICs contain high speed interfaces to the network and host memory, connected by data transfer paths.
These paths include high speed DMA engines to transfer data to and from main memory and the network. The NIC is controlled by an embedded processor which communicates with the DMA engines, network interfaces, host CPU, and on-NIC memory through a bus. Incoming network headers are copied to the NIC processor which draws them from a queue and uses them to direct the actions of the DMA engines.

For most NICs, the processor is a conventional, single-threaded, scalar processor. For these experiments, we replace this conventional processor with LWPs (Figure 4.1(b)). The LWPs are directly connected to local SRAM, and can access this memory with very low latency.

Because the LWPs trade lower IPC for reduced core complexity, and remove the necessity for a cache, they are much smaller than a conventional processor. This area reduction is based on an analysis of the PIM Lite [20, 23] processor. The PIM Lite chip contains the critical features which the LWPs proposed here utilize: wide word memory access and hardware support for multithreading. A sizing study in [167] estimates that a two LWP chip would have an area only 93% of that required by a single PowerPC 440 core. This analysis is summarized in Table 4.1. It starts with the basic 128-bit wide PIM Lite which has an area equivalent to 10.3 kilobytes of SRAM (KBe). This is then doubled to yield the 256-bit wide LWP used in these experiments. This size is doubled again to account for any additional features which a full processor core may require, and 4 kilobyte SRAM instruction cache is added. Assuming two such cores, this yields 90.4 KBe. A PowerPC 440 core consumes 33.3 KBe, however, its distance from memory requires the use of 32 KB instruction and data caches to maintain adequate performance. This totals 97.3 KBe, which is 7.6% larger than the LWP.
Figure 4.1. A high-performance network interface enhanced with (a) a processor, and (b) multiple PIMs.
TABLE 4.1

SUMMARY OF LWP CORE SIZE BASED ON COMPARISON WITH PIM LITE. UNITS ARE IN KILOBYTES (KBe) EQUIVALENTS OF DRAM

<table>
<thead>
<tr>
<th>Feature</th>
<th>Size (KBe)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base 128-bit PIM Lite</td>
<td>10.3 KBe</td>
</tr>
<tr>
<td>256-bit LWP (×2)</td>
<td>20.6 KBe</td>
</tr>
<tr>
<td>Additional features (×2)</td>
<td>41.2 KBe</td>
</tr>
<tr>
<td>4 KB SRAM Cache</td>
<td>45.2 KBe</td>
</tr>
<tr>
<td>2 LWP cores (×2)</td>
<td>90.4 KBe</td>
</tr>
<tr>
<td>PowerPC 440 Core</td>
<td>33.3 KBe</td>
</tr>
<tr>
<td>32 KB Instruction &amp; Data Caches</td>
<td>64 KBe</td>
</tr>
<tr>
<td>PowerPC Core &amp; Cache</td>
<td>97.3 KBe</td>
</tr>
</tbody>
</table>

4.2.2 Threading

The primary goal of threading is to allow multiple threads to traverse the request lists at the same time. Additionally, other message reception overhead can be overlapped with the use of threading.

For each incoming message, a commodity processor must access the header over a bus. Then, it must traverse a linked list. Finally, it must update some internal structures and attach the header data to queue (for example, moving a request to the “active” list, so it can begin a DMA). In a conventional system, the processor is only free to move onto the next header when the list traversal and setup overhead is done. In a multithreaded environment, however, one thread can be responsible for reading headers and spawning additional threads to traverse the
lists. The majority of the compute time is given to the list traversal threads. This effectively hides the latency of accessing the queue to retrieve a header, traversing the list, and performing internal accounting. Furthermore, multiple independent list traversal threads are able to hide the (already low) latency of the accesses to memory.

4.2.3 Vectorization & Wide Memory Access

LWPs have an extremely wide path between the memory and the processor core (256 bits rather than 32 or 64 bits). This path enables the processor to load large pieces of data to be compared against a single instruction. Moreover, LWPs do not have traditional caches, which typically force several sequential transfers from memory to cache (regardless of access size) and cause unneeded data to be loaded along with the requested data. By changing the data layout of the request structures, it is possible to take advantage of this wide access to memory. Aligning the comparison data into dense arrays allows up to 16 comparisons to be made with only a handful of instructions.

However, in a realistic environment, this vectorization would be of little use. This is because the dense list would quickly become fragmented and full of “holes.” To remedy this, we again turn to the LWPs multithreading capabilities. A small fraction of the processor’s processing power is devoted to a thread which occasionally traverses the lists and performs clean up and compaction operations. This maximizes the vector instructions usefulness by ensuring that the comparison buffers are “full.” This compaction thread is included in the simulation.
4.3 PIM on NIC Implementation

The MPI implementation for testing was done in three stages. First, a subset of the MPI standard was created, emphasizing NIC offload (Section 4.3.1). Secondly, this was adapted to utilize the threading aspect of LWP (Section 4.3.2). Lastly, the matching operation was vectorized and compaction was added. Several data structures changed to reflect this (Section 4.3.3).

4.3.1 MPI Subset

To explore MPI acceleration, a prototype MPI implementation has been created. This implementation is split between the host processor and the NIC processor, offloading as much MPI processing the the NIC. The prototype MPI implements a subset of MPI-1.2 [129]. With the exception of MPI_Barrier(), only basic point-to-point communication and basic support functions were implemented(Figure 4.2). Only support for basic MPI Datatypess is included and MPI_COMM_WORLD is the only group. The MPI implementation is roughly 4400 lines of C++ and is compiled with GNU g++ 3.3.

\[
\begin{align*}
\text{MPI_Comm_rank()} & \quad \text{MPI_Isend()} \\
\text{MPI_Comm_size()} & \quad \text{MPI_Recv()} \uparrow \\
\text{MPI_Finalize()} & \quad \text{MPI_Send()} \uparrow \\
\text{MPI_Init()} \uparrow & \quad \text{MPI_Wait()} \\
\text{MPI_Irecv()} & \quad \text{MPI_WaitAll()} \uparrow \\
\text{MPI_Barrier()} \uparrow & \quad \text{MPI_WaitAll()} \uparrow
\end{align*}
\]

\[\uparrow \text{functions built from other MPI functions}\]

Figure 4.2. Subset of MPI implemented
4.3.1.1 Host Software

The host portion of the MPI implementation is minimal. The three basic communication functions (MPI_Irecv(), MPI_Isend(), MPI_Wait()) offload virtually all work onto the network interface (see Section 4.3.1.2). The functions built on top of those as well as the non-communication functions are implemented on the host. To perform an MPI_Irecv(), the host checks for available space and posts an item to a queue in the network interface. The network interface manages the posted receive queue and unexpected message queue. The implementation of MPI_Isend() is similar. It only needs to place information about the message to be sent in a queue on the network interface. To implement an MPI_Wait, the host spins on a memory location in host memory. This is broadly typical of modern implementations, though some may use a blocking system call instead of a spin-lock in MPI_Wait. In either case, the host CPU performs no “useful” work during this time.

4.3.1.2 NIC-Based MPI Implementation

Most of the work is done by the processor on the NIC. It manages all NIC resources, drives the DMA engines, and handles most of the MPI semantics (including progress and matching). To accomplish this, the NIC maintains five linked lists of MPI state. This organization is similar to other MPI implementation, though others may merge the sendQ, activeRecvQ, or unexpectedActiveQ. The following lists contain requests and the state required to advance them.

- **postedRecvQ**: Posted receive buffers for incoming messages to match against
- **activeRecvQ**: Active receives requiring processing (i.e. rendezvous requests which need a reply, requests waiting for a DMA engine, etc.)
• **unexpectedQ**: List of unexpected messages. Compared to new posted receives.

• **unexpectedActiveQ**: Active unexpected messages which must be advanced (i.e. unexpected messages requiring DMA transfer).

• **sendQ**: Queue send requests for processing.

The core of the software on the network interface processor is a loop that continually checks for work. If a new message is waiting to be processed, the header is read by the NIC processor. The posted receive queue is traversed and the header is compared to each posted receive. Upon finding a match, the processor moves the receive request to the active list and either a DMA is setup (a short message) or a rendezvous reply is sent (a long message). If no match is found, the message is placed on the **unexpectedQ**, to be compared to future receives as they are posted.

New send requests cause either a DMA to start (for short messages) or a rendezvous request to be sent (for long messages). The send request is then added to the list of active requests. New receive requests are first compared to the existing unexpected message queue (**unexpectedQ**) for a match. A match causes the message to be copied appropriately or, for long messages, for the data to be requested from the remote processor. Failure to match the receive with the **unexpectedQ** will cause the receive to be placed on the **postedRecvQ**.

The processor also checks the active queues (**activeRecvQ** and **unexpectedActiveQ**) for messages that need to be advanced towards completion. Advancing messages can include such things as providing additional information to the DMA engine and responding to rendezvous replies. This enables the MPI implementation to meet the strictest interpretation of the independent progress rule while
using a rendezvous protocol and without involving a host processor thread. As items on the active queues complete, the host processor is notified by writing a location in host memory.

4.3.2 Threading Implementation

Adding threading to the MPI implementation involves two major changes.

First, the function which queried the network for incoming messages and processes them is modified. Instead of a sequential loop, the function is broken into two segments. The first is a thread which continually queries the network. Upon finding a new incoming message, it forks off a thread which performs the second segment. The second segment pulls the header from the network, traverses the posted receive queue, and then performs any internal accounting required. This organization allows the latencies of checking the network, traversing the queue, and accounting to be overlapped with other threads.

The second major change is altering the posted receive queue structure to support the fine-grained locking required for multiple parallel queue traversals. This entails adding two locks to each node, one for the pointer to the previous entry in the list and one for the next entry. When traversing the list, a thread locks both of these before modifying the node to ensure exclusivity. This allows multiple threads to iterate through the list in a pipelined manner.

4.3.3 Vector Implementation

Adding vectorization to the comparison entails two changes. The data layout for the list is modified to allow vectors of data to be loaded efficiently. Also, the comparison operation itself is changed.
Figure 4.3. (a) A conventional data layout; (b) a PIM data layout

4.3.3.1 Data Layout

LWPs have wide paths between processor and memory and provide an ALU that matches that width. Thus, the entire match (often multiple bytes) can be performed in a smaller number of instructions. Since MPI matches use relatively little data, this is a seemingly small advantage; however, a change in the organization of the list data structures enables a much more powerful usage of the wide ALU. Figure 4.3 illustrates this concept.
In Figure 4.3(a), the typical data layout of an MPI posted receive queue is shown. Memory addresses increase from left to right and from top to bottom. Although the queue entries are often sequential (due to the way list items are allocated), that has no impact on this example. If the MPI match data is 48 bits (16 bits for each of the three fields — tag, source, and communicator — is sufficient to meet the specification), then six bytes of each cache line are useful. Since cache lines are 32 (or more) bytes and processors fetch a full cache line for each cache “miss”, much of the data retrieved from memory is wasted. Sixteen memory accesses would be required to examine sixteen entries. With the data organization shown in Figure 4.3(a), the same would occur for the wide words accessed by the PIM.

The best data layout for a PIM is shown in Figure 4.3(b), where the data structures are interleaved. A single list entry has sixteen list items. All 256 bits of the PIM access to memory is used and only three memory accesses are needed to examine sixteen list items. This utilizes the wide ALU more effectively because all of the data bits are needed for matching.

In this implementation, each entry in the postedRecvQ list is comprised of four 256-bit vectors. Three of these vectors correspond to the source, tag, and communicator values for a set of posted requests. The last vector indicates which of the requests represented in the vectors are valid and which are “holes” left by previously matched entries. Each vector contains values for 16 requests.

Again, in real usage the list would quickly become fragmented and any given list entry (with 16 list items) would contain only a few valid items. In LWPs, a combination of multithreading and extremely fine-grained synchronization, can avoid this. A “list clean-up” thread traverses the list (using the low-level locking
described in Section 4.3.2) and removes “holes” that have accumulated from previous matches. Thus, at any given time, only a small amount of fragmentation exists anywhere in the list.

Though it would be possible to use this vectorization optimization in a conventional processor with a vector unit, two difficulties arise. First, conventional vector units tend to become memory starved quite rapidly – unlike LWP’s which have a greater proximity to data and can utilize the high on-chip bandwidth. Secondly, the “holes” mentioned above would quickly lead to decreased performance on a conventional implementation without a “clean-up” thread.

4.3.3.2 Comparison Operations

To perform the vectorized comparisons, the LWP assumes a basic subset of the Altivec [142] vector extensions to the PowerPC ISA. The semantics of this Altivec subset are changed to allow 256-bit vectors. Only six instructions are used (see table 4.2) and only eight vector registers are required.

The actual comparison can be broken into five steps (Figure 4.4(a-e)). First (a), the comparison values from the incoming message (source, tag, and communicator) are loaded into vectors using the vsplitw instruction. This copies the scalar value of each of the comparison values into each element of a 256-bit vector register.

Once this initial setup is complete, the comparison operation is performed on each element of the list until a match is found. The comparison (b) loads the values (source, tag, and communicator) from the list entry (which is comprised of values from 16 posted requests) into registers and compares them with the appropriate registers constructed in (a). The result of the comparisons are vectors comprised of ones if the posted request matched and zeros otherwise. The results from the
Figure 4.4. Vectorized comparison process: The incoming comparison values are loaded to registers (a), which are compared to each entry in the posted receive queue (b), the results of this comparison are combined (c), and a match is checked for (d); If found, the exact entry is found (e)
TABLE 4.2

ALTIVEC SUBSET

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>lvx</td>
<td>Load vector from memory to register</td>
</tr>
<tr>
<td>stvx</td>
<td>Store vector register to memory</td>
</tr>
<tr>
<td>vspltw</td>
<td>Copy element of a vector to all elements of another.</td>
</tr>
<tr>
<td>vcmpequh</td>
<td>Compare equal-to by half-word</td>
</tr>
<tr>
<td>vand</td>
<td>Vector bitwise AND</td>
</tr>
</tbody>
</table>

comparisons are bitwise anded together (c). This result is compared with a vector register containing information on which requests are valid and which are “holes” (d). This is done to detect if any of the posted requests have matched. If not, operations (b-d) are performed on the next set of 16 requests. If a match is detected, the exact item is selected in (e).

The selection process (e) must determine which of the 16 candidates is the matching element. It is provided with a 256-bit vector register (constructed in d) which has some bits set to one corresponding to the matching request. Simply linearly scanning each of the 16 elements of this vector could identify the match in \( O(n) \) time, but it is possible to use vector operations to achieve an \( O(\log(n)) \) search. The register is compared to a series of mask registers. This first mask determines if the match is in the first half (requests 1-8) or second half (9-16) of the register. The second mask find which quarter of the remaining register slice contains the entry. It should also be noted that a specialized instruction to find the location of the first “1” in a vector could also perform this selection in a single
### Table 4.3

**INSTRUCTION COSTS FOR VECTORIZED COMPARISON**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Figure 4.3 Labels</th>
<th>Instructions</th>
<th>Vector Inst.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>a</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>Compare</td>
<td>b–d</td>
<td>29</td>
<td>11</td>
</tr>
<tr>
<td>Select</td>
<td>e</td>
<td>24</td>
<td>12</td>
</tr>
</tbody>
</table>

The costs of this vectorized traversal are small and are summarized in Table 4.3. With these overheads, if there are 16 entries in the posted receive queue it will take 78 instructions to perform the comparison, or an average of 4.875 instructions per request. This alone is quite efficient. Considering that each request requires three values to be loaded and compared, a scalar implementation would require at least six instructions per request, and this does not include overhead for iteration, function calls, or reducing the comparisons. For a long list of 80 entries (five 256-bit registers) the vectorized approach can amortize the start up and select cost and requires only 194 instructions (2.425 instructions per request).

#### 4.3.3.3 Optimal Vector Size

To choose an optimal width for the comparison vector, a simple analysis was performed based on the performance and cost of different designs. For this analysis, we use the number of instructions in the comparison. Though the number
of cycles would be a better metric, for this first-order analysis it is reasonable to assume that the pipeline performance would be independent of vector width.

To remain competitive with the conventional “competition” it was necessary to use a design with an area less than 97.3 KBe (the silicon area of the PowerPC 440). Using the sizing estimates developed in Section 4.2.1, the largest competitive design would be a single 512-bit LWP core (86.4KBe). However, to improve throughput it is desirable to have multiple LWP cores. The widest multi-core LWP design would be two 256-bit cores.

Regardless of the size of the vector, the setup step should require the same number of instructions. The comparison is only dependent on the size of the list \(s\), and the width of the comparison vector \(w\). The selection step scales with \(O(\log_2(w))\). Using the timings found in Table 4.3, the number of instructions required can be estimated as:

\[
25 + 29\left\lceil\frac{s}{w}\right\rceil + 6\log_2(w)
\]  

Equation 4.1 is applied to various vector widths and list sizes in Table 4.4 and Figure 4.5. This analysis indicates that for lists of size greater than eight \((s > 8)\) a 256- or 512-bit vector length is desirable. When coupled with the requirements of multiple LWP cores and competitiveness with the conventional design, a 256-bit vector length is the best option.
TABLE 4.4

COMPARISON INSTRUCTIONS VS. VECTOR SIZE AND LENGTH
OF LIST s

<table>
<thead>
<tr>
<th>w = Width</th>
<th>s = 8</th>
<th>s = 16</th>
<th>s = 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>147</td>
<td>263</td>
<td>495</td>
</tr>
<tr>
<td>64</td>
<td>95</td>
<td>153</td>
<td>269</td>
</tr>
<tr>
<td>128</td>
<td><strong>72</strong></td>
<td>101</td>
<td>159</td>
</tr>
<tr>
<td>256</td>
<td>78</td>
<td><strong>78</strong></td>
<td>107</td>
</tr>
<tr>
<td>512</td>
<td>84</td>
<td>84</td>
<td><strong>84</strong></td>
</tr>
</tbody>
</table>

4.4 Experimental Setup

4.4.1 Simulated Hardware

The hardware was simulated using the simulation environment detailed in Chapter 3. The LWP-based NIC used the LWP components and the conventional NIC used a component based on the `sim-outorder` processor model. All processors used the PowerPC ISA; the LWP-based processors were also augmented with the Altivec vector extensions (discussed in Table 4.2).

The CPU was parameterized to be similar to a modern high-performance processor, such as an AMD Opteron. Although the Opteron is only six way issue, the SimpleScalar tool suite prefers powers of two and so an aggressive 8 way issue processor was modeled. Since the CPU performs little of the MPI processing, this change should not be significant. The conventional NIC processor was pa-
Figure 4.5. Comparison instructions vs. vector size \((w)\) and length of list \((s)\)

rameterized to be similar to a processor in higher-end network cards, such as the PowerPC 440 (see Table 4.5) that is used in Red Storm. For the LWP-based NIC, LWPs had a four stage pipeline, and frame caches capable of containing state for eight threads. The LWP cores used for the NIC were parameterized as described in section 2.2.6.5. The main difference is that the LWP used here is connected to a faster SRAM\(^1\), rather than DRAM memory. This is suitable for this type of embedded application, as the SRAM memory is equivalent to the SRAM cache of a conventional processor. A simple bus on the NIC connected the main processor with the DMA engine, SRAM, and matching structure. This bus was simulated with a 20 ns delay to access any components connected to it. Overall, this model attempts to provide as reasonable of a match to a real network as possible.

\(^1\)It assumes two cycles to access the SRAM.
## TABLE 4.5

PROCESSOR SIMULATION PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>CPU</th>
<th>Conv. NIC</th>
<th>PIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Q</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Issue Width</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Commit Width</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>RUU Size</td>
<td>64</td>
<td>16</td>
<td>NA</td>
</tr>
<tr>
<td>Integer Units</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Memory Ports</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>L1 (Size/Assoc.)</td>
<td>64K/2</td>
<td>32K/64</td>
<td>4K/8 (I)</td>
</tr>
<tr>
<td>L2</td>
<td>512K</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>2GHz</td>
<td>500MHz</td>
<td></td>
</tr>
<tr>
<td>Main Mem. Lat.</td>
<td>70-80 ns</td>
<td>110-120 ns</td>
<td></td>
</tr>
<tr>
<td>ISA</td>
<td>PowerPC</td>
<td>PPC/Altivec</td>
<td></td>
</tr>
<tr>
<td>Net. Wire Lat.</td>
<td>200 ns</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Net. Wire BW</td>
<td>3 GB/s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.4.2 Benchmarks

The primary motivation for this experiment was to see if we could reduce the latency of and increase the throughput for messages when long posted receive queues were present. To examine its effectiveness, a benchmark used in earlier studies [205] of queue traversal was extended.

The benchmark was originally designed to measure the impact of changes in the pre-posted receive queue length. It provides three degrees of freedom to enable the user to measure the impacts of both the receive queue length (affects caching) and of actual queue traversal (affects processing time). For this experiment, the benchmark was extended to measure the impact of long posted receive queues on message throughput. Using a traditional processor on a NIC, only one incoming message can be handled at a time. Thus, as the length of a queue grows, the number of messages that can be handled per second decreases. This shifts the standard bandwidth curve to the right — at a given message size, the bandwidth is decreased.

The message throughput can actually be decreased independently of decreasing latency. A $1\mu s$ latency that cannot be overlapped implies a limit on message throughput of one million messages per second. If that $1\mu s$ latency can be broken into five independent pipeline stages, then five million messages per second can be achieved. This approach is taken in network processors such as the Intel IXP family[80]. Alternatively, if five parallel processing units can be provided, the same effect can be achieved. To measure this effect, it is necessary to have more than one message in flight. Thus, the benchmark from [205] was modified to have multiple messages (25) in flight. The modifications to the benchmark are shown in Figure 4.6. This scenario is reasonable for applications that have
long posted receive queues, especially if they would normally communicate with multiple neighbors.

4.5 Results

The simulation of the benchmark detailed in Section 4.4.2 yielded several sets of results. First, detailed simulation of the effects of vectorization showed the validity of the matching process. Next, the effects on bandwidth and latency are considered.
Figure 4.7. Message timelines for receive-side on LWP without vectorized matching, but with threading
4.5.1 Vectors Improve Matching

Figures 4.7 and 4.8 show timelines for messages being received and processed on an LWP-enhanced NIC. Each message is shown in a separate row. For each message, four points are shown. These correspond to when the message was received (○), when the receive queue traversal begins (×), when the traversal ends (+), and when the message is marked as complete (·). These timelines were constructed from messages with 100 pre-posted receives already in the queue. The message data size was zero. Also, the “matching” time includes time to access the message header.
First, from both of these figures the effects of threading are verified. As messages arrive, they are quickly dispatched and matching begins. Also, it can be seen that message arrival, matching, and completion are all overlapping.

More importantly, the effects of vectorization are dramatically shown. The scalar comparison code (Figure 4.7) is dominated by very large matching times. For the first message, which has full access to the processor and is able to traverse the queue without blocking for others, the matching takes only 2000 cycles. However once several messages are simultaneously traversing, the processor load and locking overhead becomes significant. The matching time grows from 10416 to 22700, and then jumps to over 40000 cycles as LWPs’ frame cache becomes exhausted and it must schedule threads from memory. This explains why the slope of Figure 4.7’s completion events is so shallow.

In contrast, the vectorized matching (Figure 4.8) shows much better scaling. The first three messages take between 1056 and 1460 cycles to perform the match. Messages after that do take longer (as thread load and locking overhead grows) but even the last message takes only 18752 cycles to complete the match.

For both cases, the threading provides overlap and hides latency. However, the combination of threading and vectorization leads to dramatic performance improvement. Additionally, the use of vectorization leads to another benefit — once the match is complete, the posted receive can be removed from the posted receive queue by simply setting the “valid” bits in the vector to zero. This removes the need to dequeue the request from a list, and allows the deallocation of the request structure to be performed later. This decreases the time to complete the message from an average of 1271 cycles for the scalar case to 907 cycles for the vector case.
4.5.2 Effects on Throughput

Figure 4.9 shows a similar timeline for messages received on a conventional processor-based NIC. For the first 13 messages the match time is 2070 cycles. Messages 14-19 take 3036, and the last 6 messages take from 3538 to 3886 messages. This slow increase in matching times is probably due to cache effects. These times are still generally less than the times for the LWP-enhanced NICs. However, the critical difference is that on the conventional processor, the matches are not overlapped. Thus, while the vectorized PIM system (Figure 4.8 has started all of its matches by cycle 14000 and completed them by 33000, the conventional processor does not finish until almost cycle 76000.
Additionally, the conventional processor incurs other overheads avoided by the LWP. First, the conventional processor must dequeue the posted request, unlike the LWP which simply marks the appropriate “valid” bits. Secondly, whereas the LWP message thread can immediately begin completion, the conventional MPI must mark the message as “matched” and then later traverse the message queue to complete and deallocate the messages. Thirdly, the conventional processor must “juggle” the state of multiple requests at all times, constantly switching context to advance each message. This can be a considerable source of overhead[165].

These effects can be seen in Figure 4.10. The PIM-based solution shows dramatic improvements in bandwidth when multiple messages are in flight. Even with a short posted receive queue, the overlap achieved by having two simple processors and multiple threads with fine-grained locking can be 10-20% on short messages. Moving to longer queues rapidly exposes the advantage of the threading and vectorization and offers an order of magnitude better performance. As the length of the message grows, the time to DMA the message begins to hide the message processing time; thus, both approaches have the same asymptotic bandwidth. In general, the PIM-based approach ramps much more quickly.

4.5.3 Effects on Latency

The pure short message latency performance (Figure 4.11) is a much more mixed result. At short posted receive queues, the PIM is slower. However, these short queue lengths are not where performance problems predominate. With short queues, the processing of a single message cannot be multithreaded effectively and cannot leverage wide memory accesses or wide ALU capabilities. Furthermore, there is no significant parallelism (between multiple arriving messages) to exploit
Figure 4.10. Conventional processor-based NIC bandwidth curves (a); PIM-based bandwidth curves (b); Ratio $\frac{\text{PIM}}{\text{Conv}}$ between the two (c)
Figure 4.11. Conventional processor-based NIC latency curves (a); PIM-based latency curves (b); Ratio ($\frac{\text{Conv}}{\text{PIM}}$) between the two (c)
the capabilities of PIMS. As the queue length grows, however, PIMs are able to better utilize their unique architectural capabilities.

The relatively poor latency performance of the LWP is explained by the simplicity of the LWP processor. With only a single message, the LWP is unable to spawn multiple threads to mask latency. Profiling indicates that during the latency tests the LWP spends the majority of its cycles with only the main thread active. The operations this thread performs cannot be parallelized due to the ordering semantics of MPI and to avoid deadlock. With only a single thread, the LWP is essentially a single pipeline without branch prediction or the ability to
hide memory latency, competing against a dual-issue pipeline with a single-cycle cache. This disadvantage can negate much of the benefit of wide word comparison, especially for smaller queue lengths.

This is shown in figure 4.12 which shows a timing breakdown the single messages in for the LWP and conventional cases. This timing breakdown is for a zero length message with no other posted receives in the queue. The arrival and dispatch takes 4.2 times as long on the PIM and the matching over 6.6 times as long. Due to the differences in queue structure and threading the completion times are the same. However the overall effect is that the receive process takes almost two and a half times longer on the LWP.

However, the performance for the single message case is still reasonably close to the baseline system. Moreover, the performance for the more important case of a multiple-message system should outweigh the single message latency effects.

4.5.4 Conclusions

The problem of long message queues and high message throughput requirements are difficult to solve on conventional NIC processors due to their architectural design. Scalar, single threaded, cache-based designs show poor performance at the memory-intensive list traversals required for high throughput handling of MPI messages.

In contrast, Light Weight Processors combine low-latency access to memory, multithreading, and vector capabilities to greatly accelerate processing of messages. Although they suffer some performance loss with single message latency, they provide much better bandwidth when handling multiple messages. This improvement in bandwidth ranges from 10–20% for short queues up to an order of
magnitude performance boost for long message queues. Additionally, this can be achieved with a silicon area similar to a conventional multi-issue NIC processor.

The net effect of this is application dependent. However, it has been observed that messaging overhead has a large impact on application performance [41, 127], and the overhead of traversing long queues can be extreme[16, 19]. Because this technique greatly reduces the overhead of these queue traversals, and message processing in general, it is probable that it would cause a significant net improvement to application performance, particularly in conditions of heavy load.
CHAPTER 5

FULL/EMPTY BIT PIPELINING

The performance of parallel programs is closely related to how fast they can send, process, and receive messages [41]. The advent of increasingly capable network offload processors has allowed a large amount of this communication to be overlapped with the “real” work of computation. However, the inherent overheads associated with the most common message passing layer (MPI) leads to applications that fuse small communications into larger messages. In many cases, this forfeits some of the overlap that could otherwise be achieved.

Additionally, MPI semantics impede another potential source of overlap. Currently, a processor must wait until a message has been copied \textit{in its entirety} to main memory before the processor can do any computation on that message. This is because, for a conventional system, there is no easy way to inform an application of a partially valid buffer.

This creates a tradeoff when designing a communication pattern. On the one hand, small numbers of large messages require less overall communication overhead and impose less complexity for the programmer. On the other, large numbers of small messages may allow computation to be better overlapped with communication, but at the cost of significant computational and bandwidth overhead and program complexity. This tradeoff has been explored in [43], which found it
was difficult to find correct good parameters for segmenting messages and noted deleterious effects on code size and complexity.

Extending the standard memory semantics may offer an escape from this trade-off. One such extension, Full/Empty Bits (FEBs), adds an extra bit to each memory word. This bit designates whether the data in that word is valid (full) or not (empty). A thread which attempts to read from an “empty” word blocks until the word is set to “full.” Implemented in hardware, this can happen transparently, without requiring any modification of existing programs.

Though not limited to use in Light Weight Processors, FEBs are a complimentary technology. If synchronization meta-information is stored in the memory, having processing capability close at hand to manage and update that information can improve its capabilities. Also, having an autonomous processor which can quickly access and modify the synchronization status of memory is beneficial.

These semantics allow the construction of very fine-grained producer-consumer relationships. In this case, an incoming message buffer, initially “empty”, is gradually set to “full” by the incoming message transfer. Instead of blocking for the entire message, computation can begin immediately. If computation consumes the buffer faster than the message data can be transferred in, the FEB semantics will block the compute process. This allows computation and communication to be overlapped at a very fine grain level without requiring the computational overhead and bandwidth of using many small messages. Because computation can start more quickly, the process wastes less time in functions like `MPI_Wait()`. Reducing this overhead can have dramatic effects on application performance, especially as applications scale [127].
Figure 5.1. Typical application communication (a,b) and computation (c) pattern: Peripheral cell’s state is sent to a node’s neighbors (a), The node then waits for incoming “ghost cells” (b), and then performs the update computation (c)
For example, a typical scientific simulation code segments the simulation into a series of cells, which are distributed to each processor. Every timestep, each node sends the cells on its periphery to its neighbors (Figure 5.1(a)). It then waits for data from its neighbors which it uses to construct “ghost cells” (b). Once it has received this, it then computes the new state for each cell (c) based on the values of that cell’s neighbors (for example, the gray stencil in (c) of the target cell’s immediate neighbors).

Under a conventional system, the entire “ghost cell” buffers must arrive before any computation can begin, even though the the initial phases of the computation do not require the entire incoming buffer. With FEB pipelining, computation can begin as soon as any of the ghost cells have arrived. This removes the artificial serialization, and allows (b) and (c) to be overlapped.

This chapter explores the potential of such a system. It builds upon existing work in high-performance computing and communications. Rather than construct a radically new programming model, it looks at how the existing MPI semantics can change with the introduction of pipelined messages. We propose that the greater flexibility of pipelined messages will allow significant performance improvements without affecting the user visible environment.

A proof-of-concept implementation is discussed and simulations on micro-benchmarks and a subset of the NAS Parallel Benchmarks are used to demonstrate its effectiveness. Additionally, analysis of production applications establishes its applicability to scientific codes.
5.1 FEB Pipelining Design

Traditionally, optimizing communications patterns must trade-off between two paths. One route coalesces communication into large messages, potentially limiting overlap of communication and concurrency between processors. The other avenue breaks communications into smaller messages, allowing overlap, but incurs increased overhead. The approach of this work is to leverage extended memory semantics to transparently “pipeline” messages. This allows the user to circumvent the traditional trade-off and gain the advantage of small messages (increased overlap) while retaining the low overhead of larger messages.

With FEBs (or similar extended memory semantics) each memory word is given additional meta-information to designate the word as “full” or “empty.” If a given address is “empty,” reads to that address will block until the address is marked “full” by another thread.

In a conventional system (Figure 5.2, upper), message handling (Comm Lib) and the entire message transfer from the network (Data Trans.) must be completed before operations can begin on any portion of the buffer. Full-empty bits allow computation to be overlapped with the copying in a very fine-grained manner (Figure 5.2, lower). In such a situation, the memory words that have been filled by the communication are available immediately, while access to memory words that have not yet been filled and are still “empty” can block the application.

Because the computation performed on the data in a buffer may be complex and time consuming, the overlap of communication and data processing may allow a slower copy, a lower total execution time, more efficient communication, and simpler code. Rather than requiring a network which transfers data as fast as possible, it is only necessary to have a network which transfers data as fast as it is consumed.
5.2 FEB Pipelining Implementation

To test the feasibility of FEB pipelining, the ideas outlined in Section 5.1 were implemented in the modified subset of the Message Passing Interface (MPI) [129] mentioned in Section 4.3.1. This MPI already implements a basic subset of MPI’s blocking and non-blocking point-to-point message functions. A few more collective operations were added to the implementation, as were FORTRAN bindings. The MPI subset already demonstrates a high degree of processor offload — all of MPI’s request progress is completed on the NIC processor, freeing the main processor from these tasks. To implement FEB pipelining, three significant changes were required.

First, the `MPI_IRecv()` function was modified to set the receive buffer’s FEBs to “empty.” It was assumed that this could be performed at a high rate of speed.

Figure 5.2. Buffer transfer and consumption is serialized under a conventional system (upper), but may be overlapped with FEB pipelining (lower)
through specialized DRAM commands. These commands would perform a “bulk empty” on a range of data. Minimally, FEBs would require specialized DRAMs or memory controllers, so it is reasonable that the access mechanisms could be modified to reset several values at a time, possibly one open row at a time. If the memory were augmented with full fledged LWPs, this emptying operation could be accelerated and completed without direction from the CPU.

Secondly, the request engine was modified to mark a posted request as “completed” as soon as possible. In this case, as soon as an incoming message header is matched with a posted request. At this point the source, message size, and message tag are all available, allowing the MPI status fields to be filled in. The message request structure in main memory is marked as completed, but the request structure on the NIC is not removed. Instead, the NIC begins the DMA processing as normal, but indicates that the DMA engine should set the incoming buffer to “full” as the copy proceeds.

Thirdly, some special handling would be required if the communication semantics allow “unexpected” messages. The NICMPI implementation handles unexpected messages by copying the incoming message to a pre-allocated buffer in main memory. When the message request matching the unexpected message is posted, the NIC directs the main processor to copy from the unexpected buffer to the correct posted buffer. In this case, the main processor must also ensure that the posted receive buffer is set to “full” as the memory copy completes. In the current implementation, the DMA from the NIC to the pre-allocated buffer does not use FEB semantics, and is not pipelined.

It should be noted that the application still must check to see if a message request is “completed” in MPI_Wait(). This function still fills in the MPI_Status
structure and informs the application that the MPI_Request structure can be reused. Thus, the network latency (time to get the header to the destination) is still a factor, though the effects network bandwidth (time to get the entire message to the destination) is lessened.

The overhead for these modifications is minimal. Completing the posted request early merely shifts some communication between the NIC processor and main memory to occur earlier. The FEB adjustments for unexpected messages would also require little extra overhead as setting the FEBs can be done at the same time as the copy. The largest potential for overhead is when setting the receive buffer to “empty.” If Full/Empty bits are used, this is an $O(n)$ operation. However, the impact can be diminished in a number of ways. First, since the buffer is contiguous and the data in the buffer need not be preserved, it should be possible to instruct the DRAM, LWP, or memory controller to clear an open row at a time instead of individual words. Because an open row can be on the order of a kilobit, this could speedup the clearing operation by a factor of 32. Additionally, an LWP or memory controller could perform this bulk clearing operation without supervision from the CPU. Depending on the application, it may be possible for program to “empty” memory addresses at it reads them – sidestepping the clearing problem entirely.

No special modifications are made for error detection or correction. Because the data is still transferred over the network in the normal fashion, the normal network, link, and physical layer error handling mechanisms remain in place. MPI itself provides no error handling mechanisms to the user – the API specifies that its calls be reliable.
5.3 Experimental Setup

To demonstrate the benefits of the full-empty memory semantics in pipelining communication, several benchmarks were run inside a simulated FEB-capable system. Also, traces from several large scientific applications used at Sandia National Laboratories were analyzed. Data usage patterns were identified to evaluate if these techniques will apply to full-scale production codes.

5.3.1 Simulated Hardware

As in Chapter 4, the hardware was simulated using the simulation environment detailed in Chapter 3. All processors used the PowerPC ISA, the LWP-based processors were also augmented with the Altivec vector extensions.

The hardware setup for the NIC was similar to that in Figure 4.1(a) — a conventional processor controlling high speed DMA engines which connect the network to main memory. The NIC model contains a NIC bus, on-board memory, and DMA engine. Also, bounded internal queues for communication with the network and main processor were simulated. Main memory was modeled with a simple DRAM model which included latencies and accounted for open row (open page) hits. The system and NIC bus model modeled contention and latency.

The main processor was modeled after an Opteron-class high performance processor. The NIC processor was based on a PowerPC 440 processor, typical of high performance network cards. The nodes in the simulation were configured as described in Table 5.3.1.
### TABLE 5.1

**SIMULATOR PARAMETERS**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>CPU</th>
<th>NIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Q</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Issue Width</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Commit Width</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>RUU Size</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>Integer Units</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Memory Ports</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>L1 (Size Assoc.)</td>
<td>64K/2</td>
<td>32K/64</td>
</tr>
<tr>
<td>L2</td>
<td>512K</td>
<td>none</td>
</tr>
<tr>
<td>ISA</td>
<td>PowerPC</td>
<td></td>
</tr>
<tr>
<td>DMA Transfer rate</td>
<td>500 MB/s</td>
<td></td>
</tr>
<tr>
<td>Main memory latency</td>
<td>140-160 cyc.</td>
<td>30-32 cyc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topology</td>
</tr>
<tr>
<td>Wire Latency</td>
</tr>
<tr>
<td>Bandwidth</td>
</tr>
</tbody>
</table>
5.3.2 Benchmarks And Applications

The first benchmark is a simple micro-benchmark that performs a standard “ping-pong” operation, copying a buffer back and forth between two nodes. At each end, a simple integer averaging operation is performed on each 32-bit word of the buffer. This computation amounts to 8 instructions (including branch overhead) per 32-bit word. Timing instrumentation was added to node 0, and the timing data was broken down into three portions: Send, Wait, and Compute (Figure 5.3). To test the effects of “manually” pipelining the message, the microbenchmark can also break the message into two or four segments.

As a further proof of concept, two benchmarks from version 3.1 of the NAS Parallel Benchmarks [213] were also used. The CG benchmark performs a parallel conjugate gradient operation and the MG benchmark performs a simplified multigrid method. Other benchmarks, which required more advanced collective operations or communicators, could not be used, due to the limited nature of the MPI subset used in the simulator. These benchmarks were used to assess the impact of message pipelining on total application performance, communications and blocking time.
Trace-based analysis was also performed on three applications from Sandia: LAMMPS, CTH, and ALEGRA. These applications are detailed in Section 2.3.1.6. For these experiments, LAMMPS uses the `imp-chain` input deck and CTH uses the `cth-efp` input deck.

5.3.3 Analysis

A key metric to determine if the pipelining scheme is feasible is the rate at which data from an incoming message is consumed. If data consumption is significantly higher than the rate at which data arrives, the processor will frequently wait for data. However, if data arrives faster than it is consumed, the processor will be able to avoid blocking. Additionally, the data access pattern in the buffer is important. If buffers are accessed in a “forward” manner (“from front to back”) and the data arrives in order, the FEB scheme will be more effective than if the buffer is accessed more randomly.

To determine the data consumption rate, the Sandia applications were traced using the `amber` [6] instruction tracing tool for PowerPC. The resulting trace allowed MPI function calls to be identified, and their arguments to be collected. This allows the message buffer posted for “incoming” MPI calls (i.e. `MPI_Recv()`) to be recognized. This buffer was then tracked, and memory references to it were distinguished. This yields the number of instructions between the first and last access to the buffer. For modern processors, this approximates execution time, allowing us to determine the data consumption rate. We can also identify the buffer’s access pattern. For these experiments we classify the access pattern as either “forward”\(^1\) or “other.” Each of the applications was run for three timesteps or two billion instructions.

\(^1\)sequentially forward.
5.4 Results

The approach detailed in Section 5.3 yielded three sets of results. First, simulations of the microbenchmark provided a set of performance results detailing the operation of the FEB pipelined semantics. Secondly, simulation of the two NAS benchmarks provided insight and proof of concept as to how the approach performs in larger applications. Lastly, analysis of existing codes provided indication of how the pipelining technique could be applied to full-scale scientific applications.

5.4.1 FEB Pipelining Accelerates Application: Micobenchmark

The goal of the FEB pipelining approach was to increase message passing performance. The micro-benchmark (detailed in Section 5.3.2) was simulated to detail how this performance increase occurs.

The benchmark was instrumented to measure the time spent in `MPI_Send()`, the time spent in `MPI_Wait()`, and the time spent computing. This breakdown (Figure 5.4) reveals that the FEB pipelined approach almost completely removes blocking in the `MPI_Wait` statement, allowing computation to begin much earlier. Though the FEB-enhanced benchmark must occasionally block to receive more data, this does not seriously impact performance. For small messages, (such as the 128 byte case in the upper graph) the added overhead of splitting the message in two (Mult2) causes significant increases in all stages of the computation. For larger cases (such as the 4096 byte case in the lower graph) manually segmenting the message reduces wait time enough that the extra overhead of sending is compensated for. However, the FEB case, which reduces wait time without added overhead, performs better in both cases. Further manual segmenting of the mes-
sage (into thirds or fourths, for example) does not lead to significantly increased performance. The extra overhead of four messages outweighs the gains made by overlapping.

For smaller message sizes, “manually” pipelining the data into two messages (*mult2*) increases overhead. For longer message lengths, a two-part manual pipeline does improve performance compared to the base case, though latencies are never as low as the FEB pipelined configuration (Figure 5.5). Somewhere between 1024 and 2048 bytes, breaking the communication into two messages begins to pay
Figure 5.5. Round Trip latency over different message sizes, normalized to the base case: Results shown for manual pipelining into two (mult2) and four (mult4) messages, and for FEB off. Breaking communication into four parts has a slight pay off for larger message sizes.

FEB pipelining consistently yields lower latency across all message sizes. Small messages are accelerated by about 15% and larger messages are over 30% faster than the base case.
5.4.2 FEB Pipelining Accelerates Application: NAS Subset

The second set of simulations examined the behavior of two of the NAS Parallel Benchmarks. The goal of these simulations was to gauge the impact on how much time the application spent waiting for messages to arrive. This measurement was accomplished by having the simulator track the number of instructions spent in the wait loop of `MPI.Wait()`. Since the MPI collectives in our implementation are built on top of the point-to-point calls, this captures all waiting. Measurements were taken with and without the FEB pipelining, and for two, four, and eight nodes (Figure 5.6).

The simulations indicate that waiting was reduced substantially for all cases. Using the FEB pipelining reduced wait time in the CG benchmark by 19% to 29%, and by 25% to 33% in the MG benchmark. It should be noted that this was accomplished without any retuning or adjustment of the benchmark source code. It is probable that by restructuring communications patterns, additional performance could be gained.

Overall application performance improvements were not as large. This was due to the generally small portion of the benchmark spent in communication. The CG benchmark performance was increased by 2.7% (for two nodes) to 4.9% (for eight nodes). MG’s performance increased by 2.1% (two nodes) to 5.6% (eight nodes). However, if we assume that this performance effect continues its trend, it could amount to a rather significant improvement in performance if the problem is scaled to larger sizes.
Figure 5.6. Reduction in blocking for communications for NAS benchmarks
TABLE 5.2

APPLICATION ANALYSIS: MEAN INSTRUCTIONS PER BYTE IN BUFFER, PERCENTAGE OF BUFFERS ACCESSED IN A “FORWARD” PATTERN, AND DATA CONSUMPTION RATE IN MILLIONS OF BYTES PER SECOND.

<table>
<thead>
<tr>
<th>Application</th>
<th>Instructions per byte</th>
<th>Rate (MByte/s)</th>
<th>% “Forward”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alegra</td>
<td>6.76</td>
<td>295.9</td>
<td>79.5%</td>
</tr>
<tr>
<td>CTH: amr3</td>
<td>7.92</td>
<td>252.5</td>
<td>88.0%</td>
</tr>
<tr>
<td>CTH: efp</td>
<td>8.86</td>
<td>225.7</td>
<td>100.0%</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>4.34</td>
<td>460.8</td>
<td>59.7%</td>
</tr>
</tbody>
</table>

5.4.3 FEB Pipelining Accelerates Application: Analysis

Trace based analysis of several applications was performed to determine data consumption rates and access patterns (see Section 5.3.3). Table 5.2 shows the results of this analysis. The data consumption rate is measured by the number of instructions executed per byte of data while the buffer is being consumed (column 1). Then, assuming an IPC of 1.0 and a CPU speed of 2.0 Ghz, a data consumption rate in MBytes/sec is derived (column 3). The data consumption for these applications range from 225.7 MBytes/sec to 460.8 MBytes/sec. Though a wide range, it is below what might be expected from a modern high performance network and system bus.

Column 3 shows the percentage of buffers which are accessed in a “forward” direction — the data access pattern which would best fit the FEB pipelining scheme.
In all applications, the majority of message buffer accesses were in “forward” direction. Other than LAMMPS, over 75% of accesses would be highly suitable to FEB pipelining. Also, it should be noted that even buffers not accessed with this pattern will still gain some performance from FEB pipelining. Even in the worst case, a buffer accessed “in reverse,” would still see some overlap of the `MPI_Wait()` and computation between leaving MPI and starting buffer access.

5.5 Conclusion

Using the extended memory semantics of Full/Empty Bits allows fine-grain, low overhead producer/consumer relationships to be applied to messaging. This increases performance by allowing communication and computation to be overlapped. Though some of this enhancement can be achieved by manually breaking a message into multiple parts, this is detrimental to performance due to the added overhead and it harms program complexity.

Simulation of a microbenchmark shows in detail how this overlap may occur. Simulations of two NAS benchmarks demonstrates that these changes can be beneficial to realistic codes even without modification of the existing source. Furthermore, analysis of scientific simulations indicates that there is a significant potential for this technique to be applied to production applications.
CHAPTER 6

LOOP LEVEL THREADING

Historically, performance improvements in computer architecture have been the outcome of systematically exploiting parallelism. On the system level, increasing the number of processors has steadily increased scalability. On the level of the individual processor, techniques such as pipelining, vectorization and superscalar processing have revealed more and more instruction level parallelism. Multiprocessor systems, pipelining, vectorization, and superscalar processing have attained more and more parallelism. Improvements in fabrication technology, allowing more transistors to be placed on a chip, have enabled these enhancements. However, because these fabrication improvements have not been matched by a similar dramatic improvement in the fundamental memory architecture, a growing gap between processor and memory speed has emerged. This gap — the Memory Wall[215] — is being addressed by aggressive caching and prefetching strategies, however these techniques yield diminishing returns. A variety of novel architectures have been proposed to reduce or tolerate memory latency, however they are often wedded to new programming models which face an uphill battle towards acceptance, or are costly. For example, using large numbers of simple multi-threaded processors is impeded by the large number of single-threaded legacy codes.

The techniques of extending the efficiency of a single chip also face diminishing returns. As the amount of parallelism that can be extracted from an instruction
stream plateaus, and fabrication techniques drive the transistor budget upwards, an emerging trend is to place multiple processor cores on a single chip. Each of these cores is essentially a conventional design. Using techniques such as branch prediction, large caches, and deep superscalar pipelines each core aggressively capitalizes on instruction level parallelism. Though this approach allows increased parallelism, it does not solve the memory latency or bandwidth problems. Indeed, it has the potential to inflame it, as the bandwidth off of a chip increases more slowly than the number of cores it can contain[98].

To achieve higher performance a new architecture must reduce the impact of limited memory bandwidth and high latency while continuing to exploit multiple levels of parallelism. It must do so without hampering the execution of traditional serial codes. Additionally, it must do this without seriously increasing the overall system cost.

One such design is to augment the memory system with small processing elements. In this processing-in-memory approach, each of these light weight processors is a minimalist design, discarding features such as multiple instruction issue and branch prediction in order to minimize area and cost. The LWP's utilize their proximity to memory and low-level hardware support for multithreading to decrease the effects of memory latency. Instead of adding large conventional cores to increase parallelism, LWP's are added. A single large conventional processor is retained to direct the LWP's and to execute single-threaded code. Several LWP's can be added to each LPC chip.

This chapter explores how such a system could be used in conjunction with a well understood programming model — OpenMP[13] (Section 2.3.4) — to achieve high performance. The OpenMP model is well suited to this architecture. The
OpenMP “Slave” threads are assigned to the LWP s where they exploit loop-level parallelism identified by the OpenMP directives. The “Master” thread executes on the conventional processor where it directs the Slaves and also executes the serial sections of the program. This leverages the conventional processor’s strengths while minimizing the LWP s’ weaknesses. The serial code sections tend to access more cacheable control data and perform more branches, putting to use the large cache and branch prediction tables on the main processor. Likewise, the loop-level work units assigned to the LWP s tend to be more memory intensive and contain fewer branches.

The chapter starts with the justification for this approach. It then details the hardware and software design philosophy (Section 6.2) and their implementation (Section 6.3). It also details a set of experiments which utilize the proposed system, and their results. It also includes a brief exploration of some data management techniques which could be used to improve this technique.

6.1 Enabling Threads

This chapter seeks to take an existing type of parallelism — loop level parallelism — and find it can be better exploited by high performance systems. To do this, it must recognize and address the current impediments which prevent systems from making use loop level parallelism.

The current impediments to widespread adoption of loop level techniques fall into two broad categories: cost/performance arguments and productivity arguments.

The argument from cost / performance is that the automatic (or semi-automatic) decomposition of loops into parallel threads is too expensive, and that process
level parallelization (such as MPI) is more efficient. Much of this concern stems from the cost of large shared memory machines which are required to run parallel multithreaded codes. Other programming models, such as MPI, can utilize distributed memory architectures which tend to be less expensive and more scalable. Additionally, the mechanisms used to create and control threads in a conventional system can be quite high. For example, the code for `pthread_create()`, used to create a thread, is 259 instructions, not including subroutines and system calls. `pthread_mutex_lock()`, used for synchronization is 170 instructions. With such overheads, using threads for small tasks is difficult.

The argument from productivity is that loop-level techniques are insufficient to achieve high processor utilization. This utilization argument stems from the nature of the processors being utilized. Traditional Symmetric Multiprocessing (SMP) or Chip Multiprocessing (CMP) machines are comprised of several identical processor cores. Each of these cores is very complex and, hence, very expensive. Because of this expense, it is desirable to utilize the processing resource. Leaving it idle would imply a significant waste of resources.

The system proposed in this chapter aims defeat both arguments by reducing the cost (and improving performance) of the extra processors which make OpenMP possible. Instead of expensive cores which require herculean effort to utilize fully, cheap processors are added. This allows the extra parallelism, but at a reasonable cost.

Lowering the cost of the “extra” processors effectively lowers the barrier to entry for using threads. This is especially true if coupled with a low cost synchronization mechanism. If the cost is low, then it is worth utilizing loop-level techniques, even if there is not much effort put into them. This allows pursuing
“low hanging fruit” parallelism. For example, a number of benchmarks from the NAS Parallel Benchmark Suite (see Section 2.3.1.5) and applications used at Sandia National Labs\(^1\) were compiled “naively” — no changes to the were made to the serial version of the source code, and the compiler tried to find threadable loops. Without programmer hints, the compiler\(^2\) was conservative in choosing which loops it could thread, and only selected those which help obvious potential. Still, even with this naive compilation, a large number of parallel loops were detected. Table 6.1 shows that a significant percentage of loops were found to be parallelizable by a compiler.

### 6.2 Design

The design philosophy of the proposed system is based upon simple processors wedded to memory. The OpenMP model leverages the characteristics of both the LWPs and the conventional CPU.

\(^1\)The Sandia applications are detailed in Section 2.3.1.6
\(^2\)IBM XLF/XLC Version 8.1 for MacOS X

---

**TABLE 6.1**

**NAIVE LOOP THREADING RESULTS**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CTH</th>
<th>LAMMPS</th>
<th>ITS</th>
<th>BT</th>
<th>CG</th>
<th>FT</th>
<th>LU</th>
<th>MG</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>% of Loops Threaded</td>
<td>53%</td>
<td>39%</td>
<td>10%</td>
<td>31%</td>
<td>43%</td>
<td>55%</td>
<td>69%</td>
<td>69%</td>
<td>50%</td>
</tr>
</tbody>
</table>
Figure 6.1. The proposed LWP-enhanced architecture and a baseline CMP design

6.2.1 Hardware

Our baseline system (see figure 6.1) is a single Chip Multiprocessing (CMP) chip connected to DRAM over a bus. Each of the CMP cores is a conventional superscalar processor. A basic conventional cache coherency protocol[84] provides consistency between the processors.

This baseline is compared to our proposed LWP-enhanced system. The proposed system is comprised of a single conventional processor (configured the same as the cores in the CMP system), connected to the LWP-enhanced DRAM. One or two of the DRAM chips is replaced with a Processing-In-Memory (PIM) chip containing embedded DRAM and one or more LWP cores. Designs with many more LWP cores are feasible, however this chapter focuses on a short-term “low-impact” design which presents a minimal risk to the system cost.
6.2.2 Application Software

The software retains the syntax of OpenMP — the user annotates their code to indicate where loop level parallelism exists. This effectively breaks the program into a series of Slave threads and a Master thread. In the baseline case, the Master and Slave threads are distributed to each of the CMP cores. In the LWP case, the conventional core executes the Master thread and the Slave threads are executed on the LWPs.

6.3 Implementation

6.3.1 Hardware Implementation

The Lightweight Processor (LWP) is a simple single-issue pipelined processor. Forwarding between pipeline stages is allowed, but there is no mechanism for branch prediction. When a branch appears, the processor stops issuing further instructions from that thread until the branch is safely resolved. A four stage pipeline is modeled in this simulation, though floating point instructions are assumed to take an additional four stages, during which no other instructions are issued from the thread. This processor core is similar to the 4-way multithreaded pipeline in the recently announced Niagara processor[109] (see Section 2.2.6.5). Each LWP is attached to on-chip DRAM memory which it can access quickly. Additionally, it has exclusive access to a portion of off-chip DRAM.

The thread on the LWP can access the entire address range, however if it requires memory from outside its on-chip DRAM or exclusive off-chip area it must issue a remote instruction to the LWP which “owns” that address. The remote LWP fetches the requested memory and returns the data to the requesting LWP. During this time, the thread which has issued the remote instruction is stalled.
The remote memory instruction travels across the chip or memory bus and this may cause contention — the simulation models this. Preliminary observations indicate that 10% to 20% of accesses are remote. Each LWP has its own L1 cache and cache misses are serviced by access to local or remote memory. Remote memory accesses are not cached, removing the necessity of a coherency protocol between the LWPs. Also, due to the scope of these experiments, the communication delay for all remote memory latencies are modeled as a constant. However, contention at the remote processor is modeled.

The hardware supports thread and register state for up to eight threads. Threads are issued to the pipeline in a simple round robin fashion. This scheduling is done in a fine-grained interleaved fashion, switching between active threads every cycle. For these simulations, two Slave threads are assigned to each LWP. Incoming remote memory instructions are also scheduled in this manner, allowing them to be serviced without an interrupt.

Hardware support for low-level multithreading is built into the LWPs. This takes two forms. The first is low cost support for thread creation and automatic scheduling of the threads. The second is support for fast synchronization through Full/Empty Bits (FEB) [5, 196]. As described in Chapter 5, Full/Empty Bits append an extra bit of memory to each word in memory. This bit can be set atomically at the same time as a read or write and access to the word can be conditional upon the bit’s status. For example, a word can only be accessed if the FEB is set to the “full” position, and a read access may then set the FEB to “empty.” This mechanism is used to build mutexes, semaphores, and other synchronization machinery. This allows much faster synchronization and thread control. To improve the comparison in the experiments, it was assumed that the conventional processors were also augmented with FEB.
Because of the cost of the FEB mechanism it may be desirable to use some other mechanism (such as reservation stations) or only augment a small subset of the memory space with FEBs. However, this work focuses on FEBs due to their flexibility and simple semantics.

6.3.2 OpenMP on pthreads Implementation

The OpenMP programming model focuses on loop-level parallelism. Starting from a serial code, in C, C++, or FORTRAN, the programmer can add annotations to indicate a parallel section. Within this parallel section, loops may be annotated to show when their iteration may be executed in parallel. Additional annotations may specify which variables are held private to a thread and which are shared, and allow the user to select how iterations of the loop should be partitioned between the processors. Code not within a parallel section is executed by a “Master” thread. OpenMP can utilize a number of underlying thread systems, such as POSIX threads (pthreads) or Solaris threads.

For these experiments, an OpenMP on pthreads implementation is used. This required some modification to the pthreads library. First, the synchronization routines (pthread_mutex_* and pthread_cond_*) are modified to use full/empty bits (FEBs). Also, the thread creation routines are modified to assign the Master thread to the conventional processor and to distribute Slave threads to the LWPs. Additionally, minor changes were required to ensure thread IDs were managed correctly.

The use of FEB semantics and assumed hardware support for threading substantially simplifies the implementation of several pthreads intrinsics. For example, the code for pthread_create(), was reduced from 259 instructions to 58. Similarly, pthread_mutex_lock(), was reduced from 170 instructions to 21.
6.3.3 Work Scheduling

The OpenMP runtime library is modified to account for the heterogeneous nature of the target processors. The main focus of this modification is in the work scheduling algorithms. Early experimentation revealed that the default static scheduling was not well suited to the heterogeneous environment. Additionally, the other standard OpenMP work scheduling algorithms were not adequate to the task, so a new algorithm was devised.

6.3.3.1 OpenMP Limits

OpenMP allows several options for scheduling. There are three scheduling algorithms defined by the standard, and each of these algorithms can be parameterized. The scheduling can be specified for a given loop or for the program as a whole.

The different algorithms are:

- **static**: Loops are broken into blocks of \( n \) iterations\(^3\). These blocks are assigned to each of the threads at the start of the loop’s execution.

- **dynamic**: Loops are broken into blocks of size \( n \), like the **static** algorithm. Each thread is initially assigned one block; remaining blocks are kept in a queue. If a thread finishes its assigned block, it may claim another unfinished block from the queue.

- **guided**: Loops are broken into progressively smaller blocks until a minimum is reached. The first block contains \( \lceil \frac{\text{iterations}}{\text{number threads}} \rceil \), subsequent blocks contain \( \lceil \frac{\text{iterations remaining}}{\text{number threads}} \rceil \). Thus, if there are 100 iterations and two threads, the first

\(^3\text{Where } n \text{ is a user defined parameter.}\)
block will contain 50 iterations, the second block 25, the third 13, and so on. Each thread is initially assigned one block, and then can request additional blocks as they finish. This division continues until a user defined size $n$ is reached.

Additionally, other parties have added some non-standard algorithms:

- **affinity**(IBM)[92]: Similar to the guided algorithm. Loops are initially broken into partitions of $\left\lceil \frac{\text{iterations}}{\text{threads}} \right\rceil$. These partitions are then broken into blocks of decreasing size ($\left\lceil \frac{\text{iterations remaining}}{\text{threads}} \right\rceil$). Once a thread finishes the iterations in its partition, it can “steal” blocks from another thread.

- **affinity**(RWCP)[175]: Combined with data layout extensions to OpenMP, the RWCP’s affinity algorithm tries to schedule iterations of a loop to threads based on their affinity to a user-defined array.

Though effective for homogeneous systems, these algorithms proved inefficient for the proposed system. This occurred for two reasons. First, the default algorithm (static) does not account for one set of processors (the LWs) being much slower than another (the CPU). Secondly, the granularity by which the algorithms assign loop iterations is poor. The algorithms only take a single user defined parameter to control how loops are divided. Because each application may have several different loop sizes, this leads to loop chunks which are either too large or too small.

The standard guided and IBM’s affinity algorithms do allow a more dynamic approach. However, in the case where the difference in speed of the processors is known and roughly stable (i.e. the clock ratios of the CPU and LWs), the more dynamic approach may incur too much overhead. Testing revealed this to be the case with this experiment.
6.3.3.2 Heterogeneous Scheduling

Whenever it could be done so safely, the default work partitioning algorithm (static) is modified. Because the LWPs run at one half to one quarter of the main processor’s speed it was found that a more dynamic partitioning is required. Thus, each loop was partitioned into segments of \( \frac{N}{C} \) iterations, where \( N \) is the total number of iterations in the loop and \( C \) is the ratio of clock speeds between the main processor and the LWPs. Segments are distributed to the Master and Slave threads which begin computation. As they finish, they are allocated another segment until all segments are complete. This dynamic schedule is used for the baseline systems as well, as it gives somewhat higher performance. The constant 4 in the denominator was found by a parameter sweep to provide the best trade off between small granularity (and better load balancing) and lower overhead.

This improved dynamic scheduler takes advantage of two factors. First, the low overhead synchronization makes it possible to use lower granularity work units without the overhead of reallocating between threads harming performance. Secondly, it adapts the size of the work unit based upon the number of iterations. Rather than use a set size, as the standard OpenMP schedulers, it can create work units which are more uniform.

6.3.4 Data Management

The underlying assumption of conventional processors is that sufficient state can be kept close to the processor (via caches) to efficiently execute the program. The proposed system bypasses this by moving the sites of computation out into the memory, and using threading to hide additional latency. However, this does not assume that a given thread of execution will be close enough to the memory
it requires to execute efficiently (or that there are enough threads to hide the latencies of remote access).

To further enhance performance on the proposed system, we briefly explore three techniques to improve the data locality on the LWPs. These techniques involve caching data or moving threads, or a combination of the two.

6.3.4.1 Software Cache

The first technique is to create a software cache of data which is frequently read, but seldom (or never) written. This cache is small, and is distributed to each LWP. Because it is seldom written, this set of data can be easily cached and use a software protocol to maintain coherency. Additionally, because it is a software cache, it does not require additional expensive hardware. This sort of data analysis could be performed by a combination of compiler and profiler support, at minimal impact to the user.

The set is determined by running the application and tracing memory access patterns. Addresses are first sorted by the ratio of reads to writes and secondly by the total number of accesses. For these experiments (see Section 6.5.4), a cache of 1024 bytes was assumed. Early experiments indicated that reads to this set outnumbered writes by a factor of several hundred thousand to one, and that software cache writes comprised less than a ten millionth of the instructions committed, so the exact mechanism of resolving coherency was not modeled.

The size of the cache was chosen by analysis of the data usage patterns found in a shorter execution of the benchmark (Table 6.2). From this, an estimate of the number of times a given word would be referenced was made. Near the 1KB cache size, the number of references per word begins to fall sharply. At roughly
TABLE 6.2

SOFTWARE CACHE SIZES AND USAGE

<table>
<thead>
<tr>
<th>SW Cache</th>
<th>Est. Refs/Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 B</td>
<td>10128</td>
</tr>
<tr>
<td>256 B</td>
<td>5544</td>
</tr>
<tr>
<td>512 KB</td>
<td>3249</td>
</tr>
<tr>
<td>1 KB</td>
<td>2090</td>
</tr>
<tr>
<td>2 KB</td>
<td>1230</td>
</tr>
<tr>
<td>2.5 KB</td>
<td>Writes</td>
</tr>
</tbody>
</table>

2.5KB, the cache begins to include addresses which are frequently written. Thus, 1KB was deemed to be a reasonable tradeoff between size and performance.

6.3.4.2 Migration

The second technique takes a different tack. Rather than storing data closer to the thread, it moves the thread to the data. When a thread accesses a remote address, the thread is stopped and its state is shipped to the remote LWP which contains that data. For these purposes, the thread's state includes only its register file and program counter. To simplify matters, any uncommitted instructions from the migrating thread are squashed before migration.

Some of the implications of this technique are examined in these experiments. However, the exact mechanisms and latencies of the migration are beyond the scope of this work. A more detailed examination of migrating thread mechanisms can be found in [138] and [139].
6.3.4.3 Stack Cache

A further optimization, which can be coupled with a thread migration system, is to migrate more than just the thread’s immediate state. An obvious choice for additional state is the thread’s stack. However, absent programmer guidance, the exact range of the stack is not always known. Thus, for these experiments, we naively take 128 bytes “above” and 128 bytes “below” the frame pointer.

6.4 Experimental Setup

The experimental setup consisted of an execution based simulation of a benchmark suit. The same simulator was used for both the baseline and proposed configurations. This benchmark suite was the NAS Parallel Benchmarks.

To estimate the cost of the LWP and CMP systems, a silicon cost model was used[201]. This model considers the silicon costs of each system based on wafer processing costs and estimates of die yields. It neglects other factors such as NRE, packaging, cooling, and interconnect. The model assumes LWPs are constructed in a commodity DRAM process, but with four additional layers of metal. The LWP core is conservatively assumed to be similar to a cacheless MIPS 64K core.

6.4.1 Simulated Hardware

Our baseline system (see figure 6.1) is a single Chip Multiprocessing (CMP) chip connected to DRAM over a bus. The number of cores on the CMP is varied from one to eight. Each of the CMP cores is a conventional superscalar processor with attached L1 and L2 caches. The microarchitecture of the core is similar to an AMD Opteron Processor — it is a multiple issue pipelined processor capable of speculative execution and dynamic branch prediction. Further parameters are
TABLE 6.3

PROCESSOR SIMULATION PARAMETERS.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Conv.</th>
<th>LWP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Q</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Issue Width</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Commit Width</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>RUU Size</td>
<td>64</td>
<td>NA</td>
</tr>
<tr>
<td>Integer Units</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Memory Ports</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>L1 (Size/Assoc.)</td>
<td>64KB/2</td>
<td>4KB/8 (I)</td>
</tr>
<tr>
<td>L2 (Size/Assoc.)</td>
<td>2MB/16</td>
<td>none</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>2GHz</td>
<td>500-1000MHz</td>
</tr>
<tr>
<td>Main Mem. Lat.</td>
<td>70-80 ns</td>
<td>70-80 ns</td>
</tr>
<tr>
<td>Local DRAM Lat.</td>
<td>NA</td>
<td>5-10ns</td>
</tr>
<tr>
<td>DRAM Chips</td>
<td>8</td>
<td>8-#PIM chip</td>
</tr>
</tbody>
</table>

found in Table 6.3. A simple cache coherency protocol[84] provides consistency between the processors.

Our proposed system is comprised of a single conventional processor (configured the same as the cores in the CMP system), connected the LWP-enhanced DRAM. One or two of the DRAM chips is replaced with a Processing-In-Memory (PIM) chip containing embedded DRAM and one or more LWPs.
This hardware was simulated using the simulation environment detailed in Chapter 3. The LWP processors are represented with a simple model covering the thread control, pipeline, remote memory access, instruction cache, and EDRAM access. The DRAM model includes latencies and accounts for open row (open page) hits. The bus model models contention and latency. The model for the conventional processor cores is based on the SimpleScalar toolset’s \[28\] \texttt{sim-outorder} simulator.

The address space is distributed between the LWPs by hashing. The address range is broken into 4096 byte segments which are distributed between LWPs in a simple round robin fashion. For cases where thread migration is used, a 10 cycle latency is injected to model the migration cost.

6.4.2 Benchmarks

The benchmark suite used for these experiments is the NAS Parallel Benchmarks. The OpenMP implementation[97] is used, except for the single core baseline runs which uses the serial implementation. The “W” (workstation) problem size is used. The NAS codes are processed with the Omni OpenMP compiler (v1.6) and PowerPC GNU C compiler.

The FORTRAN and C sources for the NAS codes are processed with the Omni OpenMP compiler version 1.6, a research compiler for OpenMP. The Omni compiler produces C code with calls to the Omni runtime libraries. This code is then compiled with the PowerPC GNU C compiler\(^4\) native to MacOS X.

The applications are statically linked with a modified version of the MacOS X standard library and some support libraries, most notably the Omni OpenMP sup-

\(^4\)gcc version 4.0.0 (Apple Computer, Inc. build 5026)
port libraries, which are build atop pthreads. The Omni and pthreads libraries are modified as explained in section 6.3.2. The measurements for branches executed and cache pressure were conducted by the simulator, avoiding any instrumentation of the code.

6.5 Results

Four sets of results are presented. First, two “supporting” pieces of evidence show how branches are distributed and how cache pressure is relieved. Secondly, the primary result, how the price/performance tradeoff is effected by the LWP-based enhancements. Lastly, results related to the data management techniques are shown.

6.5.1 Branch Instruction Distribution

One goal of the OpenMP partitioning is to leverage the assets of each processor type by decreasing the number of branches executed by the LWPs. This accom-
Figure 6.3. Effect on instruction mix: LWPs receive a disproportionally low number of branches

plished by keeping the branch-heavy serial code segments on the Master thread. This leverages the CPU’s branch prediction logic. As shown in Figure 6.3, the LWPs do execute a lower number of branches than the program as a whole in all but one benchmark. In all but one benchmark, branches account for a smaller percentage of the LWP’s instruction flow than the total program. The exception, LU, may be due to the use of busy wait loops in the benchmark’s implementation.

This result is significant because it shows that OpenMP can manage the distribution of branches between the conventional and LWP processors. Depending on the LWP’s configuration, this could be important if the LWP does not include branch prediction logic, or if there are not enough threads to mask the branch latency.
Figure 6.4. The number of instructions per cache miss on the main processor is improved for the LWP setup

6.5.2 Cache Pressure

Another goal is to relieve cache pressure from the main processor. The metric adopted to measure this pressure is the number of instructions executed per data cache miss (L1 or L2). As shown in Figure 6.4, for all but one benchmark the number of instructions executed between misses\(^5\) is improved substantially, often increasing by 80%. Additionally, we see in Figure 6.5 that as we increase the number of LWPs, the number of instructions per miss continues to increase. This would follow, as the memory-oriented LWPs shoulder more of the processing.

This is significant for two reasons. First, by relieving cache pressure from the CPU’s data cache, its performance can be improved. Secondly, it shows that the LWPs are able to perform the more data-intensive parts of the computation.

\(^5\)Only L1 and L2 Data cache misses are counted
Figure 6.5. The number of instructions per cache miss continues to grow as the LWPs take more of the processing burden.

6.5.3 Silicon Cost/Performance

Figure 6.6 displays silicon cost on the X axis and performance (measured by execution time) on the Y axis (Thus, points closer to the origin are “better” since they have low execution time and low cost). Price-performance points are shown for each of the 12 systems tested (see Table 6.4). These systems are CMP systems of 1, 2, 4, and 8 cores; $\frac{1}{2}$ speed LWP systems with 1, 2, 4, and 8 LWPs; and $\frac{1}{4}$ speed LWP systems with 1, 2, 4, and 8 LWPs.

Figure 6.6 shows that the LWP configurations all have a cost somewhere between the one and two core CMP systems. For performance, two configurations are used: One in which the LWPs run at one half the CMP’s clock speed, and one in which the LWPs run at one quarter the CMP’s speed.
Figure 6.6. Price/Execution Time for LWP and CMP configurations: Si cost and execution time are normalized to a 1 core, no LWP system; The graph is cut short for clarity, so only the eight core CMP system’s execution time is shown (Si cost 3.32)
TABLE 6.4

PRICE PERFORMANCE GRAPH EXPLANATION

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (x)</td>
<td>Price-Performance point for (\frac{1}{2}) speed LWP configurations of (x) processors.</td>
</tr>
<tr>
<td>(\bigcirc) (x)</td>
<td>Price-Performance point for (\frac{1}{4}) speed LWP configurations of (x) processors.</td>
</tr>
<tr>
<td>(x) (\bullet)</td>
<td>Price-Performance point for CMP system with (x) cores.</td>
</tr>
<tr>
<td>8 (\triangleright)</td>
<td>Price-Performance point for 8-core CMP system. Due to size constraints, only the performance is shown. The normalized Si cost is 3.32.</td>
</tr>
</tbody>
</table>
The quarter-speed configurations show superior performance on three of the six benchmarks. On the BT benchmark their performance is worse than the CMP case, showing poor parallel scaling. On the LU benchmark, the quarter-speed performance is similar or slightly better than the base case and slightly worse than the two core case. For the MG benchmark, the four and eight LWP configurations perform better than the CMPs in both execution time and cost. Overall, the quarter speed LWPs win outright in three of the six cases, lose one, and “tie” the other two.

The half-speed configurations consistently show performance that is superior to the one and two core CMP systems, and at a much lower cost. Indeed, they evidence performance similar or better to the four core CMP system in four of the benchmarks, and at a much lower silicon cost. It should be noted that the LWP’s seem to scale poorly on the BT benchmark. The cause of this bears further study.

6.5.4 Data Management

The data management experiments (described in Section 6.3.4) examine six configurations for their effect on communications (Figure 6.7). These experiments were performed on the FT benchmark for the NAS suite. The FT benchmark was chosen as it performs well on the LWP architecture and its performance appears to scale well.

- **base**: Baseline unoptimized system. No migration or caching.
- **swc**: 1024 byte software cache.
- **mig**: Thread migration.
- **mig+swc**: Thread migration and software cache.
Figure 6.7. Effects of thread migration and data management

- **mig+stack**: Thread migration with 260 byte “stack cache.”

- **mig+stack+swc**: Thread migration, stack caching, and software cache.

The software data cache recorded over 5.75 million “hits,” of which only 16 were writes. This means that the average member of the cache was used over 5600 times. Combining the software cache to the unoptimized baseline configuration (Figure 6.7(swc)) decreases the number of remote memory references by 2.94%. Adding a software cache to a system with migration (mig+swc) decreased the number of thread migrations by 7.96%.

Migration drastically reduced the number of memory references (down 91.2%), leaving only a few remote accesses from CPU cache misses to be serviced.
combined number of migrations and remote references is decreased by 54.2% from the baseline case. Adding a stack cache to the migrating system (mig+stack) decreases the combined number of remote accesses and migrations by 8.9%. Combining all the optimizations (mig+stack+swc) decreases this number by 12.66%.

6.5.5 Conclusion

The combination of low cost multithreaded processors-in-memory with a conventional core shows significant benefit. In this design, LWPs reduce cache pressure on the main processor. Because of this, and their own low latency access to memory, they are able to avoid the “Memory Wall”, while exploiting substantial levels parallelism.

The OpenMP programming model efficiently partitions tasks between the two processor classes, leveraging the best qualities of both. The large conventional core is able to use its caching, branch prediction, and high serial performance. The LWPs are relieved from many of the branch instructions (which they are less effective at) and are able to use their multithreading to achieve high throughput. The adoption of a dynamic scheduling algorithm allows OpenMP critical sections to be executed in the heterogeneous setup.

All of this leads to substantial improvements in performance and cost / performance attributes. A conservative LWP design running at one quarter of the main processor’s speed was able to outperform many more expensive CMP configurations. Additionally, an aggressive design running the LWPs at one half the clock speed delivered performance superior to that of larger 2-core and even some 4 or 8 core configurations.
Finally, a cursory analysis of several data management techniques indicates that there is ample room to optimize data and thread placement. This could lead to further performance enhancements.
Parallelism exists on a number of levels. Prior chapters in this work have explored ways of improving parallelism at the process/MPI level (Chapters 4 and 5) and at the thread/loop level (Chapter 6). This chapter looks to add a new level to this hierarchy by exploiting parallelism within basic blocks. This is done by identifying very small threads, called “threadlets.”

7.1 Expanding the Hierarchy of Parallelism

To transform a modern high performance application into a form which can take advantage of the capabilities of LWPs, it is necessary to exploit parallelism on a number of levels. Additionally, to preserve the huge investment in existing codes, this transformation must be performed with minimal impact to the actual program.

7.1.1 Current Hierarchy

A single ’silver bullet’ to ensure efficient transformation of all legacy applications to an LWP architecture is unlikely, as high performance scientific computing covers a wide range of applications and programming techniques. The diversity of scientific applications already requires a diverse set of techniques to exploit parallelism where ever possible (see table 7.1.1).
TABLE 7.1

LEVELS OF PARALLELISM

<table>
<thead>
<tr>
<th>Technique</th>
<th>Parallelism</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>Process-Level</td>
<td>Application, Library</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Loop-Level</td>
<td>App. Hints, Compiler, Library</td>
</tr>
<tr>
<td>Parallelizing Compiler</td>
<td></td>
<td>Compiler, Library</td>
</tr>
<tr>
<td>Futures</td>
<td>Function Level</td>
<td>Programmer, Compiler</td>
</tr>
<tr>
<td>Vectorization</td>
<td>Short Vector</td>
<td>Compiler</td>
</tr>
<tr>
<td><strong>Threadlet Extraction</strong></td>
<td>Instruction-Level</td>
<td>Compiler</td>
</tr>
<tr>
<td>Out of Order Execution</td>
<td></td>
<td>Processor</td>
</tr>
</tbody>
</table>

Some of the techniques in this set, such as MPI and Out of Order Execution, are in widespread use today. Other techniques, such as parallelizing or vectorizing compilers, are used today, but their use is limited because conventional hardware does not effectively support them or they are only applicable to a limited set of programs. Some techniques, such as vectorization or futures require extensive program modification and are not always portable.

7.1.2 Expanded Hierarchy

New techniques must be utilized and existing techniques must be extended to maximize the performance of existing applications on LWPs. Ideally, this set of techniques can be applied with little or no impact to the application programmer.
Already Chapters 4 and 5 explore how LWP can improve the performance of existing MPI applications. This was accomplished by utilizing vectorization and low overhead multithreading. Chapter 6 examined how LWP can take advantage of OpenMP and other loop-level threading techniques. Again, this utilized the LPW’s multithreading capabilities.

This chapter adds a new level to the parallelism hierarchy: threadlets. Threadlets are very small threads which can be extracted by a compiler with no programmer involvement. They have not been utilized in conventional architectures, though they exploit parallelism similar to that of Out of Order Execution (OOE). However, by taking advantage of the LWP’s unique capabilities, they are able to avoid the expensive hardware require by OOE and achieve much greater efficiency.

Threadlets differ from other types of parallelism in several ways. First, they are implicit, unlike MPI, OpenMP or futures. They do not require programmer intervention. Secondly, the level and type of parallelism they extract is different. Figure 7.1 shows data dependency graphs of two examples, and how both loop-level techniques and threadlets extract parallelism. Figure 7.1(A) shows three independent iterations of a loop. The shaded region shows the regions which loop-level techniques may identify for parallel execution – that is, the iterations or basic blocks. Threadlet techniques identify a finer-grain level of parallelism within the basic blocks (dotted boxes in figure). Because of this, threadlet techniques can exploit parallelism which loop-level techniques often cannot. In Figure 7.1(B), several iterations of a loop are shown, but these iterations are not independent. This inter-loop dependency would probably prevent loop-level threading. However, it does not prevent the use of threadlet techniques. Thus threadlets can be used in conjunction with loop-level (and other) techniques, and they can exploit parallelism previously ignored.
Figure 7.1. Threadlets Vs. loop-level parallelism: Threadlets find parallelism within a basic block ((a) and (b)); Loop level parallelism is found over independent loop iterations (a, shaded region)
In some ways, the threadlet approach is similar to VLIW or short vector processing. All of these techniques try to exploit parallelism at a very fine level—that if the instruction—by placing the burden on the compiler. However, there are key differences. First, threadlets are more generic than VLIW or short vector in that they do not require a specific organization of data (like short vector) or a particular mix of instructions (like VLIW). Secondly, threadlets are less architecture dependent than VLIW (which assumes a fixed set of functional units) or short vector (which assumes a fixed width vector unit). In contrast, threadlets allow the scheduler to flexibly coordinate the execution of threads to maximize parallelism.

7.2 Threadlet Model

Conventional processors use complex data caches and out-of-order execution logic to eliminate latencies in instruction execution and memory access. With this, they achieve high performance, but at a significant cost in silicon area. LWPs take a lower cost approach, eliminating or hiding latencies with proximity to the memory and multithreading.

Multithreading is a natural match for the type of latency tolerance required by a LWP systems. It allows for concurrency to be discovered by the programmer and compiler, rather than adding expensive hardware. It also allows the latency of memory accesses and pipelining to be masked by the available threads, rather than by more complex hardware-oriented schemes. Most multithreaded architectures focus on a small number of relatively large threads that are used to increase total throughput, rather than to mask memory latency. For example, Intel’s Hyperthreading [126] or the multithreaded PowerPC[15] support precisely this
type of server-oriented workload. The Cray (Tera) MTA[5], on the other hand, has explored using large numbers of relatively small threads, with lighter weight creation and synchronization mechanisms.

7.2.1 Threadlets

This chapter focuses on the opportunities for parallelism in very tiny threads or threadlets with fine grain support for thread creation and synchronization. Threadlets exist within a basic block, which is a sequence of instructions occurring between two branch instructions in the program trace. This restriction serves to eliminate control dependencies, thus avoiding speculative execution or rollback concerns. A threadlet enhanced code consists of the original set of instructions (generated by a standard compiler) along with fork operations (signifying the beginning of a threadlet) and a set of synchronizations (Figure 7.2). These threadlets are well suited to be executed on a simple multithreaded processor, such as the LWPs used in this work.

7.2.2 Threadlet Synchronization

Synchronizations are required whenever one threadlet produces (Figure 7.2(b)) a value which another threadlet consumes (Figure 7.2(c)). This value may be stored to a memory location by one threadlet and then loaded by another, or it may be an integer or floating point register value computed by one threadlet and then used by another. To ensure correctness, there must be a synchronization mechanism to ensure that the consuming threadlet has some way of knowing if the producing threadlet has not yet produced the required value so it can wait for the required data. Thus, threadlets serve to extract potential concurrency from
a serial program. To some extent, this mirrors the process which a superscalar processor performs in hardware to dispatch instructions to different execution units. The main difference is that there is no central register file or broadcast bus. This it is desirable to keep inter-instruction dependencies in the same thread. By examining a program trace, we are able to quantify the opportunities for threadlet creation.

To efficiently support low-level threading, a fast hardware supported synchronization mechanism must be provided. The mechanism used here is the Full/Empty Bit (FEB), as used in the Cray MTA[31], the Sparcle [1] processors, and Chapter 5 of this document. Because this mechanism is supported by hardware, it can be performed with very low overhead.

FEBs can also be included on registers[204] allowing multiple threads to share the same register context. Sharing registers reduces the number of hardware register contexts required to support a given number of threads and it also re-
duces the costs of synchronization by not requiring data to be moved to and from memory. Shared register contexts is similar to the minithreads in the \textit{mtSMT} architecture [160] or strands in the Strand architecture[125]. Unlike the \textit{mtSMT} minithreads, threadlets use hardware mechanisms to share registers rather than having an exclusive subset of the register file. Unlike strands, threadlets are aimed at parallelism within a basic block and can be extracted by the compiler without programmer direction.

7.3 Threadlet Extraction

To generate these threadlets, the program is transformed to consist of a large master thread, which forks off small threadlets opportunistically (Figure 7.2(a)). These threads consist of less than two dozen instructions, and may share state with other threads. The actual threadlet state is extremely tiny consisting minimally of a program counter and program status word.

The program trace is split into threadlets by constructing a data dependency graph of each basic block, and applying the thread extraction algorithm given in Section 7.3.1 (Figure 7.3). Once the threads are identified, the spawn points and synchronization points are inserted into the transformed code.

![Figure 7.3. Threadlet generation process](image)

167
The goal of this process is to take an instruction flow from a serial program and transform it into a multithreaded trace in a realistic manner. For this initial experiment, traces are gathered using the amber instruction trace generator for PowerPC[6] and fed to an extraction engine and simulator via the trace-based front end of the SST (Chapter 3). A “steering” partitioning scheme[163, 217], originally developed to distribute instructions between clusters in a multicluster architecture [62], was adapted to partition the data dependency graph into separate threadlets.

This process is one which could be performed with a compiler, after instruction generation, and before assembly. The only inputs required are the instructions and the dependencies between them.

7.3.1 Algorithm

The threadlet extraction algorithm operates on a stream of instructions. It examines the data flow between instructions and assigns them into threads based on two criteria. These criteria are to balance the threads (equal numbers of instructions per thread) and to reduce synchronization between the threads.

The algorithm starts by reading in instructions and adding them to a data dependency graph (Figure 7.4(a)). When a branch is encountered, the basic block has ended, and so the graph is fed into the next stage. Next, the number of threadlets that the block will be broken into is determined (b). The number of threadlets is determined by the size of the basic block and a user defined parameter $p$. The number of threadlets is $\lfloor \frac{BB\text{size}}{p} \rfloor$. If this is one, the basic block is kept serial. Thus, the minimum size for a basic block to be threaded is $2p$ instructions. Also, $p$ is referred to as the “target thread size” since perfectly balanced threads in a sufficiently large basic block will contain $p$ instructions.
Figure 7.4. Threadlet extraction algorithm

Once the number of threads is known, each instruction is “steered” to one of the threads. This partitioning scheme attempts to minimize the need for synchronization between the threadlets, and achieve effective load balancing. Synchronization is required whenever an instruction (the consumer) requires the result of another instruction (the producer) which is in another thread. As explained in Section 7.2.2, these experiments assume the use of a low-overhead synchronization mechanism such as Full/Empty Bits. In such a scheme, synchronization itself incurs no penalty. However, if the producing instruction has not yet produced the required value, the consuming instruction must block – which does incur a penalty.

To reduce synchronization penalties, the heuristic “prefers” synchronizations between instructions which are further apart in program order. If an instruction must synchronize with another instruction which is earlier in program order, then there is a greater chance that the parent instruction will have already completed, and that the child threadlet will not have to block.

To perform the steering, each data structure representing an instruction keeps a set of “scores” for each threadlet it could be assigned to. These scores are initially set to zero and are used to determine which threadlet the instruction will
be assigned to. The score is unitless, and its value depends on a user defined weighting of the synchronization and load balancing metrics.

First, the instruction is evaluated for communication / synchronization (Figure 7.4 (c)). Each ancestor of the instruction in the data dependancy graph is checked against each candidate thread. If assigning the instruction to that thread would require a synchronization a penalty is assigned to that thread’s “score.” The severity of the penalty is dependent on the program order distance between the instruction and its ancestor. Depending on the distance $d$ and the depth of the pipeline $pipeDepth$, three cases are identified:

- $d \leq pipeDepth - 1$: If the distance is less than the pipeline depth, the instructions are very close and will probably require synchronization. Thus, a penalty of $pipeDepth$ points is assigned.

- $d > pipeDepth \times 2$: If the distance is much greater than the pipeline depth, no penalty is assigned as the ancestor will probably have completed by the time the dependent instruction issues.

- $pipeDepth \leq d \leq pipeDepth \times 2$: The “medium” case, a penalty of one point is assigned.

This procedure of subdividing synchronizations into three cases came from early experiments with small traces. The performance of these traces was observed to be very low due to synchronization effects. Further examination revealed that the initial approach of treating all dependencies as equal did not lead to effective partitions. It was necessary to discount “long distance” dependencies (i.e. $d$ is high) and place more emphasis on “close” dependencies (i.e. $d$ is low). The length
of the pipeline was found to be a good standard for where the distance cut-off should occur.

The second criteria in the partitioning scheme is load balancing. This is accomplished by modifying each thread’s score by a factor equal to its load (d). The load for a thread is given by $\frac{i}{p} \cdot weight$ where $i$ is the number of instructions already assigned to that thread, and $weight$ is a user defined parameter which indicates the relative importance of load balancing and communication penalties.

Once the score for each thread has been computed, the highest scoring thread is declared the “winner.” The target instruction is assigned to that thread. This process is performed for each instruction in the basic block.

Two examples of instruction assignment are shown in Figure 7.5. Both examples assume $p = 4$, $weight = 4$, and $pipeDepth = 4$. On the left side (I), two threads, 1 and 2, have loads of 2 and 3 respectively. This indicates that thread 1

---

**Figure 7.5. Example instruction assignment**

<table>
<thead>
<tr>
<th>Thread</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Sync.</td>
<td>-4</td>
<td>-1</td>
</tr>
<tr>
<td>Load</td>
<td>-2</td>
<td>-3</td>
</tr>
<tr>
<td>Total</td>
<td>-6</td>
<td>-4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Sync.</td>
<td>0</td>
<td>-5</td>
</tr>
<tr>
<td>Load</td>
<td>-3</td>
<td>-1</td>
</tr>
<tr>
<td>Total</td>
<td>-3</td>
<td>-6</td>
</tr>
</tbody>
</table>
has two instructions already assigned to it and thread 2 has three. The instruction under analysis is C. C is dependent on two other instructions, A and B. Instruction A has already be assigned to thread 1, so if C were assigned to thread 1 there would be no synchronization and hence no penalty. However, if C were assigned to thread 2, there would be a synchronization. A occurs seven instructions before C in the instruction ordering, so $d_{A-C} = 7$. This is the “medium” case, so a penalty of one point is assigned to thread 2’s “score.” Similarly, instruction B is already assigned to thread 2. If C were assigned to thread 1, there would be a synchronization, so a penalty is assigned to thread 1’s score. instruction B is two instructions before C ($d_{B-C} = 2$). This is less than the pipeline length ($pipeDepth = 4$) so a penalty of $pipeDepth$ is assigned to thread 1. As defined earlier, the load penalties of each thread are $\frac{1}{p} weight$, which is $-2$ for thread 1 and $-3$ for thread 2. The total scores are thus $-6$ for thread 1 and $-4$ for thread 2. Hence, the instruction C is assigned to thread 2.

On the right side of the figure (Figure 7.5(II)), is a similar case. Here the loads for thread 1 and 2 are 3 and 1 respectively. Also, both of the ancestor instructions A and B are already assigned to thread 1. Thus, no synchronization penalty is assigned to thread 1, and a penalty of $-5$ is assigned to thread 2. After factoring the load penalties, the total scores are $-3$ for thread 1 and $-6$ for thread 2. Thus, in this second example, instruction C would be assigned to thread 2.

Once the basic block has each of its instructions assigned, the threadlets performance is simulated.

It is possible to weight the communication and load scores to place emphasis one or the other. For these experiments, they were weighted to count both equally.

Figure 7.6 shows an example of threadlet execution based on example I (right
Figure 7.6. Example instruction execution

side of Figure 7.5). In this example, instruction A is assigned to thread 1 and instructions B and C are assigned to thread 2. Instruction C requires operands from instructions A and B. Additionally, A occurs well before B or C in the instruction flow. Though there may be other instructions executing in the pipeline, they are ignored for clarity.

As the example starts, A begins executing in the pipeline, finishing in cycle 4. Also in cycle 4, B starts execution. C begins shortly after B (cycle 6). Quite possibly, another thread issued an instruction in cycle 5. B and C continue executing. In cycle 7, B is completing just as C requires its operands. C requires one operand from A (which has already completed) and one from B. Because B is completing, it is able to forward the needed result back to C. Thus, C can complete without delay in cycle 9.
7.4 Experimental Setup

7.4.1 Simulated Hardware

Two hardware configurations were tested: a sample LWP pipeline and a conventional processor.

The LWP was modeled as a four stage pipelined multithreaded processor. Floating point instructions required an additional four cycles to complete and memory instructions an additional six cycles. Thread spawning was assumed to take one instruction, but after that instruction executed it would be another four cycles before the new thread could begin issuing. Like the LWP described in Section 6.3.1, there is no branch prediction.

To model synchronization, it is assumed that Full/Empty Bits or some other low-cost mechanism is used. Thus, if a data item is available the instruction does not block or require any additional latency. Similarly, if the data item is not available, the instruction is squashed and the thread is stalled until it is available. This may require flushing the pipeline of other instructions from the same thread to keep execution in-order.

To give a baseline performance result, a conventional processor was modeled after an Opteron Core (see Table 6.3). However, since we are interested primarily in the performance of the processing pipeline and not the memory and other systems, the conventional processor was given perfect branch prediction and cache behavior.
7.4.2 Benchmarks

Traces were gathered from a number of scientific applications. These traces were four billion instructions in size. The applications used are detailed in Section 2.3.1.6.

These benchmarks were chosen as representative set of scientific applications. One notable characteristic is their large basic block size. Other than Cube3 and zChaff, all of the benchmarks have an arithmetic mean basic block size of over 10 instructions (Figure 7.7).

Recalling that the minimum size for a basic block to be threaded is $2p$ instructions (where $p$ is the user-defined target thread size) Figure 7.8 demonstrates how this large basic block size leads to large sections of threadable code. As $p$ was swept from 16 to 4, the percentage of instructions which were in threadable loops increased. The mean percentage for the benchmarks ranged from 35% ($p = 16$) to over 65% ($p = 4$). If the three benchmarks with small basic blocks (cube3-cbs, cube3-vbr, and zchaff) are not included, this ranges from 52% to 90%.
Figure 7.8. Threadable instruction vs. target thread size
7.5 Results

7.5.1 Threadlet Characteristics

The principle threadlet characteristics are those relating to the load balancing and synchronization between threads. The load balance can be characterized by the difference in the number of instructions assigned to each of the threads. We define the imbalance for a thread \( t \) as 

\[
\text{size}_t - \text{size}_{\text{avg}}
\]

where \( \text{size}_t \) is the number of instructions in the thread and \( \text{size}_{\text{avg}} \) is the average thread size.

Figure 7.9 shows the mean thread imbalance for each of the applications and for three different values of \( p \). For all of the benchmarks, the mean imbalance was less than 14%. This indicates a high level of balance between threads. For most of the applications, as \( p \) is made smaller the imbalance increases. \texttt{zchaff} was an exception to this, probably due to the small sample size at \( p = 16 \).

The next threadlet characteristic relates to the synchronization requirements. This is measured by the number of remote references made per thread. For this measurement, a remote references is a register value which is computed in one thread and then used by an instruction in another thread. This gives some indication of the synchronization requirements for the application. If more remote references are made, more synchronization must be done. However, it does not give a full picture because it does not capture the distance between those synchronizations. If an instruction creates a value which is not needed for quite some time, and the threads in question share a register set, there may be little or no overhead to the remote reference.

The number of remote references per thread is shown in Figure 7.10. This number is quite dependent on the target thread size (\( p \)) - larger threads require more synchronization. Overall, the number of references \textbf{per instruction} remains
Figure 7.9. Mean percentage imbalance
Figure 7.10. Remote references per thread
quite modest. Other than LAMMPS, most threadlets would require roughly one synchronization every five instructions. LAMMPS does require more synchronization, but still a manageable amount - roughly once every three instructions.

7.5.2 Performance: IPC

To examine performance, the IPC of threadlet code is compared against the baseline (no threadlets) code and the same instructions executed on a conventional 4-way processor (Figure 7.11). The mean IPC for the benchmarks on an LWP without threadlets is 0.29 (i.e. running a single thread leaves about $\frac{3}{4}$ of the pipeline empty, on average.) Adding threadlets raises this substantially (Table 7.2). Using threadlet extraction with $p = 4$ provides the best performance, with an IPC 159% that of the base (no threadlet) case.

Executed on a conventional processor, the benchmarks all achieve an IPC between 0.8 and 1.25, with a mean of 1.02. By comparison, the LWP’s best IPC is a lethargic 0.46 ($p = 4$). However, when the relative size and complexity of the processors is considered, the LWP is far more efficient.

<table>
<thead>
<tr>
<th>$p$</th>
<th>16</th>
<th>8</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>vs. no Threadlets</td>
<td>121%</td>
<td>137%</td>
<td>159%</td>
</tr>
<tr>
<td>vs. 4-way Processor</td>
<td>33%</td>
<td>39%</td>
<td>45%</td>
</tr>
</tbody>
</table>

TABLE 7.2

MEAN THREADLET PERFORMANCE (IPC)
Figure 7.11. IPC
7.6 Conclusions

Serial code, especially the large basic block code found in scientific applications, can be easily broken into small threadlets. This can be done with a simple “steering” algorithm which partitions instructions between threads based on optimizing load balancing and reducing synchronization requirements.

These threadlets take advantage of several aspects of the LWP architecture - namely, low overhead thread creation and synchronization. Augmenting a register file with Full/Empty Bit synchronization and allowing multiple threads to share a set of registers further reduces the cost of synchronization.

The use of threadlets can substantially improve the IPC on the simple LWP pipeline. This can offset some of the disadvantages of the simple LWP design when compared to a conventional processor, and without requiring additional hardware or programmer modification of the original serial code. This technique would be particularly important in a system comprised only of LWPs.
CHAPTER 8

CONCLUSIONS

The scientific computing community must decide what the next-generation and next-next-generation supercomputer will look like and how to program it. This work focuses on these community goals in the context of Light Weight Processing (LWP) — specifically, how the combination of processing-in-memory, hardware multithreading, short vector processing, and low level synchronization in memory can be used to construct feasible supercomputers. To prove feasibility, these future computers must execute traditional “dusty deck” programming models efficiently, provide improved performance by circumventing the “memory wall” and address the fundamental long-term issues in high performance computing.

8.1 Contributions

To that end, this dissertation presents contributions in two major areas: MPI (Chapters 4 and 5) and threading (Chapters 6 and 7).

8.1.1 MPI and LWP

Chapter 4 explores how LWP can accelerate NIC-based MPI processing. An MPI implementation was adjusted to use wide-word operations and multithreading to accelerate its queue traversals. The combination of multithreading and
fine-grained synchronization allows multiple queue traversals to be overlapped. The wide-word operations greatly accelerate the matching procedure.

The simple LWP processor does incur somewhat higher single message latency, but it provides much better bandwidth when handling multiple messages. This improvement in bandwidth ranges from 10%–20% for short messages up to an order of magnitude for long message queues – where performance improvement is especially needed.

8.1.2 Full/Empty Bit Pipelining

Chapter 5 uses the extended memory semantics provided by full/empty bits to turn MPI messages into tight producer-consumer relationships by “pipelining” the message transfers. This allows the receiving/consuming side of an MPI transfer to begin using data as soon as it arrives, rather than waiting for the entire buffer to arrive. A proof of concept MPI using the FEB pipelining is constructed and tested on a micro-benchmark and a subset of the NAS Parallel Benchmarks. Also, trace analysis of scientific applications is performed to find the applicability of this technique to realistic workloads.

The microbenchmark results reveal that computation and communication is more effectively overlapped, eliminating much of the time spent waiting for data. This provides substantial improvements in latency (15%–30%) over the non-pipelined case and over “manually” pipelining messages.

The NAS subset shows that the benchmark applications do have a substantial portion of their MPI related wait time reduced. This reduction occurs in all cases and is accomplished without modification to the benchmark source code.
Finally, the trace analysis shows two things. First, the rates at which applications consume data are below what might be expected from a modern high performance network and system bus. This indicates that real applications could use FEB pipelining without losing efficiency. Secondly, the majority of buffers are consumed in a “forward” pattern which is optimal for FEB pipelining. These findings confirm that FEB pipelining is applicable to realistic scientific applications.

8.1.3 Loop Level Threading

Chapter 6 augments a conventional processor with LWPs in the memory system. This system is coupled with the OpenMP programming model. The OpenMP partitioning leverages the high serial performance of the conventional processor and the abundant parallelism and proximity to memory of the LWPs. The addition of an improved heterogeneous scheduling algorithm improves the execution efficiency. Analysis of the silicon area of the LWPs indicates a minimal additional cost.

Substantial improvements are seen in both performance and cost/performance when compared to CMP designs. On the NAS Parallel Benchmarks, an LWP-enhanced system with LWPs running at one quarter of the main processor’s speed was able to outperform many more expensive CMP configurations. Additionally, an a design with the LWPs at one half the clock speed performed better than larger 2-core and even some 4 or 8 core CMP configurations.

A preliminary analysis of several data management techniques, such as thread migration and different caching schemes, indicates a high potential for optimizing data and thread placement.
8.1.4 Threadlets

Chapter 7 explores a new level on the parallelism hierarchy: threadlets. These threadlets are very small (dozens of instructions or less) threads that are extracted from basic blocks by the compiler without programmer involvement. They allow the simple LWP processor to increase its “single thread” performance without the complicated and expensive mechanisms used in conventional Out of Order Execution.

Chapter 7 presents a “steering” algorithm to partition instructions between threads. This partitioning is performed to maximize parallelism and minimize the amount of synchronization between threads. When applied to several traces from scientific applications, the algorithm substantially improves the IPC of a simple LWP pipeline. This offsets the disadvantages of the simple LWP design when compared to a conventional processor.

8.2 Cumulative Performance

Each of the techniques presented in this work are valuable by themselves, but they do lead us to ask what the future could hold if we can combine the enhancements optimally. Estimating the performance impact of these techniques can be done in several ways. For simplicity, an estimate is constructed by finding a possible range of performance improvements for each of the techniques and then multiplying them together.

8.2.1 MPI Techniques

Estimating the performance impact of the MPI related techniques involves extrapolating the experimental results to realistic applications and also to larger
sizes. Though not as accurate as actual simulation of full applications at size, this should provide a reasonable first order approximation.

8.2.1.1 LWP for MPI

Chapter 4 presents techniques to improve MPI message processing performance. To estimate the overall performance impact, the LogP network application model [41] was applied to the applications analyzed in [127]. The LogP model provides performance estimates for parallel applications based on four factors:

- **L** (Latency) : The delay in communicating a small message from its source to its destination. This is limited to the “wire delay” not the total latency.

- **o** (Overhead) : Length of time a processor spends transmitting or receiving a message. This is closest to the message processing overhead, which is addressed in Chapter 4.

- **g** (Gap) : Minimum interval between consecutive messages – this is the time it takes for a message to cross the bandwidth bottleneck in the system. This, the bandwidth available to a processor is $\frac{1}{g}$ messages per unit time.

- **P** (Processors) : The number of processors. For these estimates, the application characteristics for a 32-node system were used.

Through application analysis and experimentation, the LogP model derives a sensitivity relationship for changes to overhead. This relationship allows the runtime of an application to be predicted based on changes in the overhead ($\Delta o$), the number of messages sent ($m$) and the original runtime ($r_{orig}$), with Equation 8.1:

$$r_{pred} = r_{orig} + 2m\Delta o$$ (8.1)
TABLE 8.1
PERFORMANCE IMPACT OF DECREASED OVERHEAD

<table>
<thead>
<tr>
<th>Application</th>
<th>$r_{\text{orig}}$</th>
<th>$m \times 10^6$</th>
<th>$\Delta o = -0.20o$</th>
<th>$\Delta o = -0.57o$</th>
</tr>
</thead>
<tbody>
<tr>
<td>radix</td>
<td>7.8</td>
<td>1.279</td>
<td>123.5%</td>
<td>218.4%</td>
</tr>
<tr>
<td>EM3d</td>
<td>38</td>
<td>4.765</td>
<td>117.0%</td>
<td>170.8%</td>
</tr>
<tr>
<td>sample</td>
<td>13.2</td>
<td>1.294</td>
<td>112.8%</td>
<td>147.9%</td>
</tr>
<tr>
<td>em3d</td>
<td>114</td>
<td>8.316</td>
<td>109.2%</td>
<td>131.8%</td>
</tr>
<tr>
<td>Mean</td>
<td></td>
<td></td>
<td>116%</td>
<td>167%</td>
</tr>
</tbody>
</table>

With Equation 8.1 and the application data from [127], the performance improvements for several applications can be computed. In Table 8.1, five applications are presented. Their original runtime ($r_{\text{orig}}$) is given in seconds (column 2), and the number of messages sent per processor is shown (column 3). Columns 4 and 5 show the performance impact based on a 20% reduction in the overhead ($o$) and a 57% reduction. These numbers correspond to the message throughput improvements presented in Figure 4.10 for short queues and long 100 entry queues. The result is a mean performance multiplier between 116% and 167%.

8.2.1.2 FEB Pipelining

Chapter 5 presented a technique for pipelining messages using Full/Empty Bits. Simulation showed the performance effects on two applications from the NAS Parallel Benchmark suite, but only scaled up to 8 nodes. To estimate the
First, using the assumption of strong scaling, the compute time ($comp_{orig}$) and communication time ($comm_{orig}$) for a 32-node system was calculated for both of the benchmarks. These times were then normalized to sum to 1.0. Next, based on the observed reductions in blocking time shown in Figure 5.6, we assumed a 33% reduction in communication time. Thus, the predicted communication time ($comm_{pred}$) was 33% lower than the original communication times. Since we assume the computation time remains the same, the total predicted run time ($total_{pred}$) can be computed from Equation 8.2.

$$total_{pred} = comm_{pred} + comp_{orig}$$  \hspace{1cm} (8.2)

As shown in Table 8.2, this leads to an improvement multiplier of 1.11 to 1.20.

### 8.2.2 Thread Techniques

Calculating the performance effects of the two thread techniques is more straightforward. For the technique of combining OpenMP on an LWP-enhanced hetero-
geneous system (Chapter 6), we calculate the performance improvement over a single CPU conventional system. Assuming the LWP run at \( \frac{1}{2} \) the clock speed of the CPU, we use the mean performance of the benchmark applications as our performance multiplier. To calculate the minimum improvement we use the 2-LWP core case, and to calculate the maximum improvement we use the 8-LWP core case. This yields a range between 2.29 and 5.06.

The improvement yielded by threadlets (Chapter 7) is taken for the smallest thread extraction parameter runs \((p = 4)\). For this size of threadlet, a mean performance improvement for an application is 1.59 (Table 7.2). Depending on the application, the improvement multiplier ranged between 1.20 and 2.15.

8.2.3 Cumulative Effect

Multiplying together these effects can yield the potential cumulative effect on performance (Table 8.3). These estimates indicate that is combined optimally, the techniques presented in this work could boost performance between 3.54\( \times \) to a staggering 21.8\( \times \). Put into perspective, this magnitude of performance increase would be equivalent to jumping ahead 2.7 to 6.67 years on Moore’s Law. Though these estimates are only first order, they do indicate a high upward potential and call for more exploration.

8.3 LWP Features

The technique of Light Weight Processing consists of four features: Processing-in-Memory, short vectors, hardware support for multithreading, and extended memory semantics. As explained in section 2.2.6, these features support and
TABLE 8.3

ESTIMATED CUMULATIVE EFFECT ON PERFORMANCE

<table>
<thead>
<tr>
<th></th>
<th>Perf. Multiplier</th>
<th>Estimation Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWP for MPI</td>
<td>1.16</td>
<td>1.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LogP, scaled to 32-nodes</td>
</tr>
<tr>
<td>FEB Pipelining</td>
<td>1.11</td>
<td>1.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Const. Comm. Scaling</td>
</tr>
<tr>
<td>OpenMP for LWP</td>
<td>2.29</td>
<td>5.06</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1/2 speed 2-8 LWPs</td>
</tr>
<tr>
<td>Threadlets</td>
<td>1.20</td>
<td>2.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IPC Improvement</td>
</tr>
<tr>
<td>Total Improvement</td>
<td>3.54</td>
<td>21.80</td>
</tr>
<tr>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>Years</td>
<td>2.7</td>
<td>6.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Moore’s Law</td>
</tr>
</tbody>
</table>

reinforce each other. The contributions presented in this work utilize these features (or a subset thereof) in different ways and to different extents (see Table 8.4).

Some techniques, such as accelerating MPI with LWP (Chapter 4), rely on all of the facets of LWP. FEB Pipelining (Chapter 5), only relies on the extended memory semantics, though this may be best implemented by moving some additional capabilities into the memory. Loop Level threading techniques (Chapter 6) reply primarily on the hardware multithreading and synchronization, but use those features to exploit the proximity to memory provided by PIM. Threadlets (Chapter 7) focus only on the hardware multithreading, but require some form of low-cost synchronization (possibly only to registers) to function efficiently.

These varying feature requirements show two things. First, that the features of LWP are applicable to a range of techniques. Secondly, it shows that an incremental approach is possible. Even if all of the component technologies of LWP are not available, individual parts are still valuable.
8.4 Where Do We Go From Here?

These contributions show the viability of Light Weight Processing as a technology. Actual widespread adoption of LWP depends on a number of economic and social factors and, perhaps most of all, chance. However, it is possible to chart a potential path for the future of LWP.

This section explores such a path in three stages. Each of these stages looks at what a future supercomputer would look like. First, possible near-term adoption strategies (Section 8.4.1) are explored. Then, we look at where LWP may appear once it is a well accepted technology (Section 8.4.2). Lastly, a bold look at the next-next-generation of supercomputers and how the lessons learned from LWP will apply (Section 8.4.3).

8.4.1 Near-Term: Low-Risk

Outstanding issues related to LWPs (such as OS interaction, the best programming model, ISAs, etc...) will cause them to be viewed with suspicion. Ex-

---

TABLE 8.4

LWP FEATURE USAGE

<table>
<thead>
<tr>
<th>Technique</th>
<th>Chap.</th>
<th>PIM</th>
<th>Short Vec.</th>
<th>HW MT</th>
<th>Ext. Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI for LWP</td>
<td>4</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>FEB Pipeline</td>
<td>5</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop Level</td>
<td>6</td>
<td>√</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Threadlets</td>
<td>7</td>
<td></td>
<td>√</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>

---
isting scientific applications, dominated by FORTRAN and MPI, will not wish to undergo massive rewrites to fully utilize LWP technology. Compilers, debuggers, and optimization tools geared towards LWP will lag behind the “mainstream” tool chains. More importantly, programmers who are well versed in parallel threading techniques will still be a rare commodity.

Near-term adoption strategies will have to emphasize low risk. First, LWPs can appear in system components which do not have to support legacy code or programming styles. A prime example would be in the NIC processor (as explored in Chapter 4) where they could improve existing applications without requiring rewrites. Secondly, LWPs can be used as low-risk/low-cost accelerators in machines which appear more or less conventional. This would allow existing programming models (such as OpenMP, explored in Chapter 6) to be used. Also, existing codes could be incrementally modified to utilize the LWPs. To reduce risk, these accelerator LWPs will be sparse, with only a handful per chip to reduce cost. At this time compilers will begin to adapt by adding optimizations catering to LWP technology, such as threadlets (Chapter 7).

These incremental changes to existing architectures will allow LWP a foothold into mainstream computing. By keeping costs low and retaining existing programming models, the barrier to entry is lowered and further adoption is encouraged. As LWPs become more accepted, more resources (both hardware and programmers) will be devoted to them. This will also lead a period of trial and error as new programming models emerge and interfaces are standardized.
8.4.2 LWPs As Accepted Technology

At some point, LWPs will be an accepted technology. The continued growth of the “Memory Wall” will continue to impair conventional cache-based processors and drive the bulk of processing resources closer to the memory. Advances in fabrication technology may allow the LWP processors to adopt some features of conventional processors (e.g. out-of-order execution, parallel pipelines) though they will still remain less complex than the conventional processors of today. At some point, “PIMs” will cease to be referred to as PIMs and simply be called “processors.” Memory with extended semantics will be standard and required for the now common place massively multithreaded applications.

Current “legacy” applications will continue to exist, but will place a greater emphasis on the levels of parallelism best expressed with LWPs. Older optimizations designed to improve cache efficiency or fine-tune message passing for “dumb” memories will remain in codes and will baffle future programmers – much like codes optimized for vector machines then ported to scalar processors contain strange data layouts which confuse current programmers. Identifying parallelism and managing data placement will be key aspects of programs. Tools such as compilers, active profilers, and run-time systems will seek to improve productivity and will leverage the abundant processing power in LWPs to do so.

8.4.3 Post-Silicon: Lessons Learned

At some point, the era of silicon-based computing will come to a close. Quantum effects, increased fabrication costs, and thermal dissipation requirements will become such hinderances to silicon computers that a fundamentally new technol-
ogy will emerge. Possible contenders are Quantum-dot Cellular Automata (QCA), carbon nanotubes, or other nano-technology.

The shape and characteristics of these next-next-generation computers is difficult to predict. However, we can reasonably assume two trends will continue in importance: locality and parallelism. Due to the inherent dictates of the speed of light, moving data will continue to be an expensive operation. Intelligent data placement and caching strategies will become more important, but joining the processor and memory will be required. Parallelism will also continue to grow. This will require a much greater number of processors. Also, because the processors will have to be integrated with memory, they will have to be physically small so that memory density can be preserved.

New languages, libraries, and programming models will arise which treat locality and parallelism as fundamental parts of programming. This will be as expected of future languages as features like structured goto-less programming or type checking are expected today. The complexity of computation will require parallelism be extracted on multiple levels, using multiple techniques, both implicit and explicit. The programmer, compiler, and runtime will all take an active role in both data management and parallelism.

In short, we may not know what the next-next-generation of supercomputers will be, but they will look like Light Weight Processors.

8.5 Conclusion

Future generations of supercomputers will have to improve performance of “dusty deck” programs by a variety of techniques. These techniques must improve parallelism, circumvent the memory wall, and still cost-effective. Light Weight
Processing provides a feasible approach to achieving those goals. By combining proximity to memory and multithreading, it is able to reinvigorate the execution of current programming models and address many of the long-term issues in high performance computing.

8.6 Future Work

Future work will follow a number of directions. The results already presented can be extended in a number of ways:

- **MPI and LWP**: Exploration of the tradeoffs between different vector widths and instructions. This can better determine the class of instructions which should be supported (e.g. a $O(1)$ search instruction) and the most efficient vector width.

- **MPI and LWP**: Single thread performance improvement. Improving the performance of single threaded execution could improve the poor single-message short-queue latency of the messages.

- **FEB Pipelining**: Exploration of FEB pipelining the message sends. Although it may require extensions of the MPI syntax, it may be possible to apply the technique of pipelining to outgoing buffers.

- **FEB Pipelining**: Circumvent MPI_Wait(). If the source and tag of an incoming message (i.e. information provided by the MPI_Status record) are not required or are already known, it should be possible to circumvent MPI_Wait() all together. This would provide further latency reduction.
• **Loop Level Threading**: Better data management. Extending the techniques presented briefly in Section 6.3.4 could increase performance and scalability substantially.

• **Loop Level Threading**: Exploration of naive threading. Explore the impact of naive multithreading (Table 6.1) when combined with LWP. Due to the low cost of LWPs, even the minor parallelism which can be extracted naively from existing programs may be beneficial.

• **Threadlets**: Optimization of thread extraction algorithm. The basic thread extraction algorithm could be optimized. Experimentation with the relative weightings of the communication and load penalties could be useful for fine-tuning the extraction to the particulars of an architecture.

• **Threadlets**: Another application of threadlets would be to improve parallelism between multiple LWP cores. Instead of just hiding pipeline latency, the threadlets could be executed on different LWPs, improving performance.

Also, the simulation infrastructure (Chapter 3) could be developed further:

• **Parallelization**: To truly explore the scalability of parallel systems, adding support for parallel execution of the simulator will be necessary.

• **OS**: Support for an operating system will add additional accuracy to the simulator, as well as improving the range of applications which can be simulated.

• **New Components**: Adding new components is always a useful way to explore new optimizations. More detailed memory and bus components,
specialized hardware, and different implementations of existing components would all yield increased detail.

In addition to expanding the existing research, there are a number of other possibilities LWP usage. LWP should allow improvements to the “surface/volume” tradeoffs in MPI systems. The extra processing power may allow more effective run-time monitoring. Also, the new capabilities and requirements of LWP may lead to new programming models and design patterns.

8.6.1 MPI Tradeoffs

The increased number of processors in an LWP system should allow current MPI codes to be partitioned between processing elements in different manners. Where as conventional MPI assigns one processor to each MPI process, an LWP enabled MPI can choose a number of assignment ratios between an MPI process and LWP or LPCs (Figure 8.1). A more fine grained partition may exploit more parallelism, but at a greater communication cost. A coarse grain partitioning may sacrifice parallelism for lowered communication. This makes the traditional “surface/volume” tradeoff more flexible. The effect of this should be analyzed to estimate how much extra parallelism can be extracted from MPI codes when they are ported to LWP systems.

8.6.2 Run-Time Feedback

The “extra” processing power in an LWP-based or LWP-enhanced system would lend itself to increased introspection. A minimal proof-of-concept runtime monitor should be examined. This system would attempt to identify performance bottlenecks and communicate their existence to the programmer during program
Figure 8.1. Conventional MPIs (a) limit the ratio of MPI process to physical processors; LWP MPIs can use a wider range of assignments execution. This data could be used in a manner similar to a profiler, or it could be used during program execution to improve performance.

The runtime monitor will interface with key hardware or OS data structures such as LWP frame queues, parcel buffer handler, and cache statistics for the EDRAM. Data from these will be recorded and analyzed to produce recommendations to the program, or to take actions such as migrating data or changing its hashing scheme (figure 8.2).

A runtime system could enhance the optimization process by providing useful feedback to the compiler and programmer and improve performance.

8.6.3 LWP Design Patterns

Rather than focus on a new language and compiler, identifying commonly used metaphors or patterns should highlight how to best utilize LWPs in future systems. The recognition and implementation of key design patterns[161] for LWP systems should be a major goal of future work.
### TABLE 8.5

POTENTIAL BOTTLENECKS DETECTABLE BY A RUNTIME MONITOR

<table>
<thead>
<tr>
<th>Bottleneck</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Hotspot</td>
<td>Frequently accessed region of memory</td>
</tr>
<tr>
<td>Processor Overload</td>
<td>LWP/LPC with more threads than it can efficiently handle</td>
</tr>
<tr>
<td>Cache Thrashing</td>
<td>Memory access pattern which reduces EDRAM cache effectiveness</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Regions of Code/Memory which cause many threads to block</td>
</tr>
<tr>
<td>Excess Communication</td>
<td>Regions of code which generate communication</td>
</tr>
</tbody>
</table>

Figure 8.2. The runtime monitor analyzes current conditions and suggests actions
### Table 8.6

#### Potential Design Patterns

<table>
<thead>
<tr>
<th>Problem</th>
<th>Context</th>
<th>Forces</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Latency</td>
<td>Iterative numeric loop, traveling threads</td>
<td>Data may be highly distributed</td>
<td>Divide loop into gather &amp; compute phases to aid migration</td>
</tr>
<tr>
<td></td>
<td>Large buffer copies or transforms</td>
<td>Other operations are waiting on data arrival</td>
<td>Pipelined Copy</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Producer / Consumer relationship (small fan-out)</td>
<td>P/C relationship defines data flow. May indicate parallelism. Must have synchronization.</td>
<td>“Producer-Push”</td>
</tr>
<tr>
<td></td>
<td>Producer / Consumer relationship</td>
<td></td>
<td>“Consumer-Pull”</td>
</tr>
<tr>
<td>Locality &amp; Load Balancing</td>
<td>Complex / changing data relationships</td>
<td>Data layout cannot be easily determined, but is critical to performance</td>
<td>Tag objects for active runtime support</td>
</tr>
</tbody>
</table>

Some potential design patterns (table 8.6.3) have already been identified, and should be explored further. These patterns fall into one of several categories of problems which arise in LWP computation, such as dealing with memory latency, synchronization, and load balancing. The contexts for the patterns are situations
frequently found in high performance scientific computing.

The goal of these patterns is an elegant expression of a reusable effective solution. The emphasis should be on incremental and modular patterns, rather than attempting to introduce a completely new programming model.
REFERENCES


166. A. Rodrigues, R. Murphy, P. Kogge, and K. Underwood. Characterizing a new class of threads in scientific applications for high end supercomputers. In 18th International Conference on Supercomputing, St. Malo, France, June 2004. ACM/SIGARCH. To Be Published.


