IMPLEMENTING SCALABLE LOCKS AND BARRIERS ON LARGE-SCALE LIGHT-WEIGHT MULTITHREADED SYSTEMS

A Thesis

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

Master of Science

by

Srinivas Sridharan, B.E.

_____________________
Peter M. Kogge, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
July 2006
IMPLEMENTING SCALABLE LOCKS AND BARRIERS ON LARGE-SCALE LIGHT-WEIGHT MULTITHREADED SYSTEMS

Abstract

by

Srinivas Sridharan

Improvements in fabrication technology, guided by Moore’s law, has provided significant boost in clock rate and hence performance of microprocessors. Over the years, large amounts of chip area have been dedicated to components that try to extract more parallelism from single instruction streams. However, as we scale the process technology to smaller feature sizes, traditional techniques of exploiting Instruction Level Parallelism (ILP) especially using superscalar processors has yielded diminishing returns in terms of cost/performance.

This thesis focuses on computer architectures, which increase the opportunities for concurrency usually not possible in systems based on complex superscalar based cores. We are specifically interested in the LWP architecture that supports light-weight multithreading capability coupled on the same die as the memory by using the Processing-In-Memory (PIM) technology or the embedded DRAM technology.

The primary objective of this thesis is to explore scalable synchronization mechanisms for LWP architecture. We explore the design space for efficient low overhead implementations of mutex and barrier implementations that scale as the number of threads increases. We try to achieve this with a combination of hardware and software techniques depending on the target requirements.
Towards this goal, we attempt to answer some of the following questions in this thesis:

- How best to implement current synchronization mechanisms in the LWP architecture?

- What sort of additional hardware or software support should be added to the LWP architecture to enhance the implementation?

- How does the LWP implementations compare to that on current architectures/ISAs?

- Are there any new techniques that can be developed for LWP architecture?

- Are any of these LWP based ideas applicable in conventional systems?
Dedicated to

His Holiness Poojya Sri Sri Muralidhara Swamigal

and

Lord Sri Kalyana Srinivasa Perumal (Malaipettu, Sriperumbudur)
# CONTENTS

**FIGURES** ........................................................................................................ vi

**TABLES** ........................................................................................................ viii

**ACKNOWLEDGMENTS** .................................................................................. ix

**CHAPTER 1: INTRODUCTION** ........................................................................ 1
  1.1 Overall problem ......................................................................................... 1
  1.2 Scalable Parallel Architectures ................................................................. 4
  1.3 Highly multithreaded LWPs ................................................................. 7
     1.3.1 LIMA and Pole-vaulting the Memory Wall .................................... 7
     1.3.2 LWP Architectures ....................................................................... 8
     1.3.3 Importance of synchronization in LWP ....................................... 10
  1.4 Objective .................................................................................................. 11
  1.5 Approach .................................................................................................. 12
     1.5.1 Scalable synchronization mechanisms for LWPs ..................... 12
     1.5.2 Active Graph data structures .................................................... 13
     1.5.3 Thread migration to improve cache utilization ......................... 13
     1.5.4 Benchmark Suite ...................................................................... 14
  1.6 Thesis Organization ............................................................................ 14

**CHAPTER 2: BACKGROUND WORK** .............................................................. 15
  2.1 Processor-In-Memory Systems .............................................................. 15
     2.1.1 Rationale behind Processor-In-Memory systems ................... 15
     2.1.2 Technology viewpoint .................................................................. 17
     2.1.3 Examples of next generation PIM Systems ............................. 18
  2.2 Multithreaded and Multicore Architectures ......................................... 20
     2.2.1 Multithreaded Architectures ..................................................... 20
     2.2.2 Multicore Architectures ............................................................ 21
  2.3 Synchronization Mechanisms ................................................................. 23
     2.3.1 Waiting algorithm ....................................................................... 25
     2.3.2 Example: PThreads synchronization API ............................... 26
     2.3.3 Example: OpenMP synchronization ........................................ 28
  2.4 Barrier Synchronization Algorithms .................................................. 28
2.4.1 Hardware barrier synchronization ........................................ 29  
2.4.2 Centralized barriers ...................................................... 29  
2.4.3 Combining tree barriers ................................................ 30  
2.4.4 Dissemination barriers .................................................. 31  
2.4.5 Tournament barrier ...................................................... 32  
2.4.6 Tree barrier .............................................................. 33  
2.4.7 Adaptive and Fuzzy approaches ....................................... 34  
2.5 Scalable Mutex Synchronization algorithms ................................ 35  
2.5.1 Test-and-Set and Test-and-Test-and-Set locks .................... 35  
2.5.2 Ticket lock ............................................................. 38  
2.5.3 QOLB Hardware queue Lock ......................................... 39  
2.5.4 MCS, CLH: Software Queue based algorithms .................... 40  
2.6 Evaluation of synchronization primitives .............................. 40  

CHAPTER 3: APPLICATION BENCHMARK SUITE ............................. 42  
3.1 Microbenchmarks .......................................................... 42  
3.2 SPLASH2 Application Suite ............................................. 44  
3.3 NAS Parallel Benchmarks ................................................ 46  

CHAPTER 4: PROPOSED LWP SYSTEM ARCHITECTURE .................. 50  
4.1 Proposed System Architecture .......................................... 52  
4.1.1 LWP Core ........................................................... 53  
4.1.2 LWP Memory system .................................................. 56  
4.1.3 Thread ............................................................... 60  
4.1.4 LPC Architecture ..................................................... 61  
4.1.5 Overall system organization ....................................... 64  
4.2 Single thread consistency model ...................................... 66  
4.2.1 Single thread Consistency Model ................................ 67  
4.2.2 Enforcing ordering using STE/STAM instructions ............... 73  
4.2.3 Compiler-view of instruction ordering ............................ 76  
4.3 Summary ........................................................................ 78  

CHAPTER 5: SCALABLE SYNCHRONIZATION FOR LWP SYSTEMS ..... 79  
5.1 Using OpenMP to explore Heterogeneous architectures .......... 80  
5.1.1 LWP based Heterogeneous CMP design ............................ 80  
5.1.2 OpenMP programming model ...................................... 82  
5.2 All-LWP system ............................................................ 83  
5.3 Structural Simulator Toolkit (SST) ..................................... 84  
5.3.1 Frontends ............................................................. 85  
5.3.2 Backends .............................................................. 86  
5.3.3 Framework ........................................................... 86  
5.4 Scalable Synchronization Mechanisms ................................ 86  
5.5 SST Simulation Methodology ............................................. 88  
5.5.1 Baseline CMP and SMP System ................................... 88  
5.5.2 Simulated Architecture setup and parameters ................. 89  
5.5.3 Runtime Library Compilation Process ............................. 91
FIGURES

2.1 Four threads in a Dissemination barrier ......................... 31
2.2 Eight threads in a Tournament barrier .......................... 32
2.3 Eight threads in a Tree barrier ................................. 33
2.4 Dissecting a barrier synchronization point ...................... 41

4.1 Simplified view of an LWP core ................................. 53
4.2 Dword: 65-bit long memory word ............................... 56
4.3 Blocked Threads Frames ......................................... 61
4.4 Simplified view of an LPC ....................................... 62
4.5 Parcel Format .................................................... 64
4.6 Possible LWP-EMU mappings ...................................... 65
4.7 Simplified view of an LWP system ................................ 66
4.8 Compiler’s view of instruction ordering .......................... 77

5.1 Baseline CMP System ............................................. 81
5.2 Proposed LWP based Heterogeneous Processor System ............ 81
5.3 Baseline SMP System ............................................. 83
5.4 Homogeneous All-LWP system .................................... 84
5.5 Software stack and simulator components ........................ 88
5.6 Barrier Microbenchmark: All-LWP and SMP ....................... 96
5.7 Barrier Microbenchmark: All-LWP and SMP ....................... 98
5.8 Lock Microbenchmarks: SMP ..................................... 99
5.9 Lock Microbenchmarks: All-LWP (1 LPC with 8 LWPs) .......... 101
5.10 SPLASH2: All-LWP (1 LPC with 8 LWPs) ....................... 103
5.11 SPLASH2: All-LWP (1 LPC with 8 LWPs) ................................. 104
5.12 Lock Microbenchmarks: SMP ........................................... 105
5.13 Microbenchmarks: Heterogeneous LWP system (4/8 LPC with 8 LWPs each) ..................................................... 106
5.14 MG: Total Exec. time .................................................. 107

6.1 Active graphs ....................................................... 110
6.2 Active Graph Barrier .................................................. 116
6.3 SALT/DimC: Benchmark compilation process ..................... 119
6.4 SALT/DimC: Barrier microbenchmark .............................. 122

7.1 Migration flow within the Scheduler .................................. 131
7.2 Microbenchmarks: % Reduction in number of CPU cycles .... 135
7.3 Microbenchmarks: % improvement in L2 miss ratios .............. 136
7.4 Microbenchmarks: % improvement in L3 miss ratios .............. 136
7.5 Microbenchmarks: % Reduction in number of bus requests .... 137
7.6 Splash2: % Reduction in number of CPU cycles .................. 138
7.7 Splash2: % improvement in L2 miss ratios ....................... 139
7.8 Splash2: % improvement in L3 miss ratios ....................... 139
7.9 Splash2: % Reduction in number of bus requests ............... 140
TABLES

1.1 COMPARISON OF LWP, NIAGARA AND ELDORADO PROCESSORS ........................................ 8
2.1 PTHREADS SYNCHRONIZATION API ................................................................. 27
2.2 OPENMP SYNCHRONIZATION DIRECTIVES ..................................................... 28
3.1 SPLASH2: BASE PROBLEM SIZE AND COMPLEXITY ........................................ 47
4.1 EXTENDED MEMORY STATE DESCRIPTIONS .................................................. 57
4.2 DESCRIPTION OF LOAD/STORE INSTRUCTIONS ........................................... 60
4.3 DESCRIPTIONS OF SYMBOLS USED IN TABLE 4.2.1 ........................................ 68
4.4 INITIAL EXTENDED MEMORY STATES FOR LOCATIONS M1 AND M2 ................... 69
4.5 ORDERING OF TWO INSTRUCTION PAIRS .................................................... 70
4.6 MEMORY SEQUENCES WITH $M1 = M2$ AND $r1 = r2$ ...................................... 71
4.7 DATA DEPENDENCY HAZARDS ........................................................................... 73
4.8 MEMORY SEQUENCES WITH $M1 = M2$ AND $r1 \neq r2$ ...................................... 74
4.9 ENFORCING ORDERING WITH STE/STAM INSTRUCTIONS ............................ 76
5.1 PROCESSOR SIMULATION PARAMETERS ......................................................... 91
5.2 SPLASH2: SIMULATED PROBLEM SIZE .......................................................... 93
6.1 PARTIAL LIST OF DimC LANGUAGE EXTENSIONS ........................................... 112
6.2 SALT LWP SIMULATION PARAMETERS .......................................................... 118
6.3 ACTIVE BARRIERS: INST. MIX, AVG. AND MAX. THREADS ............................... 121
7.1 BASELINE SYSTEM PARAMETERS ................................................................. 132
ACKNOWLEDGMENTS

This work is based in part upon work supported by the Defense Advanced Research Projects Agency (DARPA) under its Contract No. NBCH3039003. The Itanium2 servers were provided by a grant from HP with additional support from NSF (CNS0447671, IIS0515674).

I would also like to thank my advisor, Dr. Peter Kogge for his continued motivation, guidance and support. He encouraged me to pursue my ideas on synchronization protocols and a number of related topics. I am grateful for the tremendous time, energy, and wisdom that he invested in steering my research. He has been an excellent role model through his dedication to quality research.

I would also like to thank Dr. Jay Brockman and Dr. Doug Thain for agreeing to serve on my thesis committee and providing useful insights.

A number of people have helped me with simulation tools, ideas, clarified a number of doubts. My research experience has been enriched through collaborations with:

- Dr. Arun Rodrigues for developing and supporting SST and also for the work on OpenMP.
- Brett Keck, Dr. Richard Murphy and Dr. Surendra Chandra for collaborating on the OS project.
- From Caltech JPL, Paul Springer and Gary Block for developing and supporting SALT/DimC.
• Branden Moore and Kyle Wheeler for clarifying a myriad of issues with simulations, *NIX and what not.

• Shannon Kuntz for SALT, DimC and multithreaded models.

• Shyam Kumar for his discussions on DRAM/EDRAM and computer architecture.

I would also like to acknowledge Prof. Venkateswaran (WARFT, India), for playing an important role in mentoring me early on during my undergrad years. Finally, I would like to thank my father J. Sridharan, my mother Girija Sridharan and my sister Anusha Sridharan for their constant support and motivation.
CHAPTER 1

INTRODUCTION

The primary objective of this thesis is to explore scalable synchronization mechanisms for architectures that support light-weight multithreading capability coupled with a balanced memory subsystem (LWP and PIM-based architectures). We explore the design space for efficient low overhead implementations of mutex and barrier implementations that scale as the number of threads increases. We try achieve this with a combination of hardware and software techniques depending on the target requirements.

1.1 Overall problem

Improvements in fabrication technology, guided by Moore’s law, has provided significant boost in clock rate and hence performance of microprocessors. Computer architects have tried to exploit this by aggressively pursuing parallelism on both system and instruction levels. Some of these include multiprocessor systems, vector processing, cluster based systems, pipelining and superscalar processing.

Over the years, large amounts of chip area have been dedicated to components that try to extract more parallelism from single instruction streams. Adding larger on-chip caches, increasing issue width, improving branch prediction depth/accuracy and speculative execution has been synonymous with each new processor generation. However, as we scale the process technology to smaller feature sizes, traditional tech-
niques of exploiting Instruction Level Parallelism (ILP) especially using superscalar processors has yielded diminishing returns in terms of cost/performance.

The problem is in some ways the result of the now commonly called “Memory Wall” problem. This is a direct side effect of the difference in commercial DRAM and logic fabrication processes. The primary design goal of DRAM fabs has been to cram in larger capacity (in terms of bits per unit area) with each successive generation. Not surprisingly, current predictions of the Moore’s law is driven by DRAM capacity. The DRAM fabs have been optimized for increasing the capacity and traditionally this has been achieved at the cost of higher latency (in processor cycles) of access. In other words the increase in memory capacity over the years, has come at the increased latency of individual memory load/store operations.

In contrast to DRAM manufacturing trends, processor logic fabs are optimized for building faster logic in order to achieve higher clock rates. This has created a imbalance in the system performance due to the disparity between the speeds at which the processor is able to generate requests and the latency required by the memory to satisfy those requests. This phenomenon has been observed in von-neumann architectures for more than a decade and is aptly named the “Memory Wall”.

Traditionally, architects have attempted to address the memory wall problem by adding large caches to the processor, increasing the size of the load/store buffers etc. These attempts along with techniques such as speculative execution are yielding diminishing returns because of the limits in parallelism available from single instruction streams and the ever increasing latency of memory accesses. Additionally the power consumption of these complex hardware components have been exponentially increasing.
Unfortunately all this has translated to a increasing gap between the theoretical peak performance and actual delivered performance. For example, the average performance of the top ten supercomputers in the Top500 List is around adds up to 75% of the peak performance of those machines. These number are based on well tuned LINPACK benchmarks that are highly parallel. However when we consider the effects of real world applications, programming effort, maintenance cost etc. the real productivity of these machines greatly diminishes.

The industry and academia are currently pursuing the idea of “multicore” chips to solve the above issues and to use the chip real estate more effectively. This idea involves placing multiple processing cores on a single chip. This allows more parallelism to be exploited at the processor level. By clocking the individual cores at lower clock rates, this also addresses some of the power consumption issues. However, as in their single core counterparts, a large percentage of on-chip transistors is still spent in on-chip caches.

The idea of multicore chips is not new. EXECUBE[1] was arguably the first design to integrate multiple processing cores on a single chip. Currently a number of designs such as Cray Eldorado[2], Sun Niagara[3] and IBM Power5[4] have successfully integrated two to eight cores in a single chip. Not surprisingly the next generation gaming consoles from Sony[5], Microsoft [6] and Nintendo all use multicore chips.

Unfortunately, multicore chips do not solve the memory wall problem. In reality, the latency gap between processing cores and memory is exacerbated by multicore chips. Now instead of having one processor core stall on pending memory requests, we have multiple cores stalling on memory. Further, there is also higher contention for bandwidth as there are more requests to be satisfied. Applying all these ideas to a large scale high performance system adds a completely new dimension.
1.2 Scalable Parallel Architectures

Parallel high performance machines have made significant advances in size, design methodology and implementations over the years. What makes designing large scale machines more complex nowadays is the fact that current HPC applications demand more than just raw processing power. In contrast to most commercial applications, HPC applications tend to use very large data sets and complex data access patterns taxing the memory system. Further large scale parallel applications require frequent synchronization amongst the processing entities (processors, threads) and this can be very time or resource consuming as their number increases. All this stress the need for higher bandwidth and lower latency of the interconnect fabric and the memory system.

The future of high performance computing inevitably includes one involving large number of software components such as threads/processes accessing an equally large number of physical resources such as processors, memory etc all operating in parallel. The big question is how to design, build and program these highly scalable machines and effectively make available to the software what is available in hardware.

A number of competing approaches are being proposed to tackle this problem. The end application performance depends on a large number of factors which includes raw processor performance, memory performance, inter-processor communication performance and I/O performance. However a balanced HPC system must provide equal importance to each of these factors in-order to support a wide range of applications.

The mechanisms used to support communication among individual processing entities in a parallel system constitutes the communication abstraction[7]. The communication abstraction is the set of mechanisms that (either in software or in hardware or both) enable the parallel entities to communicate during program execution.
Two of the most widely used parallel communication abstraction are message passing and shared-address space abstraction. The Message-passing abstraction provide each processing entity (primarily processes) with a local memory that is accessible only to that entity and require them to communicate through explicit messages. In contrast, systems with a single shared address space make the entire memory accessible to all processors/threads and allow them to communicate directly through read and write operations to memory.

The term shared memory is loosely used to refer to both architectures and programming models that support a single address space. Systems such as Stanford DASH[8], MIT Alewife[9] and Cray MTA are shared memory architectures, whereas multithreading libraries such as POSIX threads provide a shared address space are examples of programming models. The important detail to remember here is that memory accesses are not dependent on the location of the data. This provides a single address space view of the entire memory to individual processing entities. This implies memory accesses to physically remote memory are made possible through hardware or software techniques.

The key advantage of single address space abstraction is that it greatly enhances programmability of a very large system. In contrast to message-passing abstraction, the ability of each processing entity to access all the available memory simplifies programming by reducing the need for explicit data partitioning and data movement. Further, the single address space abstraction provides better support for parallelizing compilers and standard operating systems. All these factors enable the programmer to develop and tune parallel applications with substantial ease.

On shared memory architectures (and programming models) processes/threads communicate by sharing data structures. Based on the granularity of the data shared, shared memory access can be classified into fine-grain and coarse-grain
shared memory. Traditionally, fine-grain sharing is allowed when there is hardware support for shared memory, while coarse grain sharing is more commonly seen in software DSM systems. In the latter case, the shared memory abstraction is implemented in software with little or no hardware support hence is not as efficient in terms of performance. Examples of machines that support fine grain sharing includes Stanford DASH, MIT Alewife etc, and examples of software DSM systems allowing coarse grain sharing include Shasta, Treadmarks and Ivy.

Consistency of shared data structures is ensured by using synchronization constructs that protect the data from conflicting operations. These constructs play a very important role in the fine-grain sharing due to the frequency of synchronization in applications. Locks and barriers are the most popular synchronization constructs used to coordinate threads in any parallel/multi-threaded system. Locks are primarily used to provide mutual exclusion whereas barriers are extensively used to synchronize threads across different phases of a computation.

As the size of the system in terms of number of processing entities increases, depending on the application, the frequency of synchronization also increases. As the system scales to hundreds of thousands of processors the cost of the individual synchronization operations have to be kept as small as possible to achieve good application performance. Scalable synchronization protocols primarily mean that with the increase in the number of participating threads/processors the overheads involved in synchronization is kept as small as possible.

In this thesis we pursue some non-traditional approaches towards solving some of the above problems. The primary design goal is to tackle the issue of memory wall and reduce overheads of synchronization in large scale machines. The following sections enumerate some specific problems and also provide some details on how we plan to solve them.
1.3 Highly multithreaded LWPs

1.3.1 LIMA and Pole-vaulting the Memory Wall

This thesis focuses on computer architectures which increase the opportunities for concurrency usually not available to systems based on complex superscalar based cores. Specifically we explore the architectural realm encompassing two ideas developed here at Notre Dame, namely:

**LIMA** or Lightweight Multithreaded Architecture [10]. The class of LIMA based systems combine a number of software, ISA and micro-architectural improvements to exploit concurrency within the context of large scale shared memory systems. Some of the capabilities include light weight thread management, direct thread to thread communication, locality aware thread execution and extended memory state which in addition to a thread state provides novel execution capabilities.

**PIM** or Processing-In-Memory (Merged Logic and DRAM (MLD), Embedded DRAM (eDRAM) and Intelligent RAM (IRAM)) [1][11][12] technology. This is primarily a semiconductor fabrication technique that allows significant amounts of processing logic to be placed along with high density memory components on a single chip. This provides processing components access to a high bandwidth, low latency memory hence tackling the memory wall problem.

The full strengths of the LIMA architecture can be exploited if it is coupled with a high bandwidth and low latency memory system. The combination of LIMA and PIM based systems provide a wide range of opportunities to express and exploit parallelism in many levels.
### TABLE 1.1

COMPARISON OF LWP, NIAGARA AND ELDORADO PROCESSORS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>LWP Architecture</th>
<th>SUN Niagara</th>
<th>Cray Eldorado</th>
</tr>
</thead>
<tbody>
<tr>
<td># cores per chip</td>
<td>8–16</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Clock Speed (MHz)</td>
<td>500–1000</td>
<td>1000-1200</td>
<td>500</td>
</tr>
<tr>
<td>Instruction issue</td>
<td>In-order</td>
<td>In-order</td>
<td>In-order</td>
</tr>
<tr>
<td>Pipeline stages</td>
<td>4</td>
<td>6</td>
<td>21</td>
</tr>
<tr>
<td>Pipeline forwarding</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Hardware threads</td>
<td>8–16 (frame cache)</td>
<td>4</td>
<td>128</td>
</tr>
<tr>
<td>Thread switch</td>
<td>interleaved</td>
<td>interleaved</td>
<td>interleaved</td>
</tr>
<tr>
<td>Speculative issue</td>
<td>No</td>
<td>Only for memory operations</td>
<td>No</td>
</tr>
<tr>
<td>L1 ICache</td>
<td>4KB/8-way</td>
<td>16KB/4-way</td>
<td>64KB/4-way</td>
</tr>
<tr>
<td>L1 Dcache (per core)</td>
<td>none</td>
<td>8KB/4-way</td>
<td>none</td>
</tr>
<tr>
<td>L2 Dcache / Buffer</td>
<td>100MB on-chip</td>
<td>3MB/12-way</td>
<td>128KB/4-way</td>
</tr>
<tr>
<td>Cache coherency</td>
<td>shared EDRAM</td>
<td>shared L2 cache</td>
<td>on-chip buffer</td>
</tr>
<tr>
<td>Outstanding memory operations</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Limited by data dependency</td>
<td>8–entry store buffer per thread</td>
<td>one per thread</td>
</tr>
</tbody>
</table>

#### 1.3.2 LWP Architectures

The underlying philosophy of LIMA based architectures is to enable building large scale systems from simple yet highly efficient processing cores. A preferred environment for such architectures would be in systems built from technologies such as PIM. This would allow integrating multiple CPU cores and high bandwidth, low latency memory on the same chip. In this thesis we term such a chip as a Lightweight Processing Chip (LPC) and each processing core as a Light Weight Processor (LWP). A LPC will then consists internally of a large collection of LWPs and Embedded Memory Units (EMUs).\(^1\)

\(^1\)The terminology is borrowed from [10].
Each LWP is highly multithreaded, allowing a very large number of threads to be concurrently in execution within both the LWP and the larger LPC. This is possible because of the light weight nature of the threads executing on them. These threads are called Light-Weight Threads (LWTs) signifying that they own minimal state and thus are easy and efficient to manage. The table 1.3.2 presents the configurations of some comparable “light weight multithreaded” architectures such as SUN Niagara [3] and Cray Eldorado [2].

An LWP based system will consist of a very large number of LPCs connected together with high bandwidth low latency network. The LWP hardware implementation unlike most modern architectures, does not include any of the complex structures such as instruction reorder buffers, branch predictors, write buffers, per-processor counters etc. Part of the reason is the diminishing returns of such complex functional units for large scale systems. Further given the small state of the LWTs, providing “common” microarchitectural features across LPCs or LWPs is probably too expensive.

The LWP architecture allows light-weight processing capabilities on the same die as the memory by using the PIM technology or the embedded DRAM technology. The EMUs are primarily DRAM memory macros with additional capabilities of supporting rich memory semantics. This is achieved using adding extra bits to each memory word and using the state of those bits to determine how the memory word is interpreted. More specifically each memory word (64-bits) has an additional bit called the full/empty bit (FEB). If the bit is set (or full), it means that the memory word contains data whereas when unset (an empty) represents the presence of metadata rather than data itself. The concept of meta-data is very useful in representing the status of the operations on the memory word and helps determine the best way to satisfy the current request.
The system we consider here supports shared memory abstraction in hardware and involves additional mechanisms for extensions into software. It should be noted that there is no caching of remote memory locations and the need for hardware cache coherency protocols. The shared memory abstraction aids programmability of the system and the absence of hardware support of cache coherency keeps the implementation simple. However the architecture does have additional features to ensure that the cost of remote operations are not too prohibitive. Further a rich set of ISA extensions enable techniques such as low cost thread creation/join, moving the thread state to the location of remote data, fast producer-consumer synchronization using full/empty bits and atomic operations at memory.

1.3.3 Importance of synchronization in LWP

The LWP architecture is targeted towards building large machines with hundreds of thousands of LPCs supporting a large number of threads (around a million LWTs). Given the philosophy of the architecture it is of very important to support low cost fine-grained synchronization. Depending on the application and programming model characteristics, as the number of threads increases the cost and frequency of synchronization also increases. Further the end-to-end latency of memory or network events across such a system is also quite high for a large system and increases with higher contention. Thus it becomes important to explore synchronizations mechanisms that uses the key features of the LWP architecture and scale with the size of the target machine.

Currently, the LWP architecture provides low cost support for single producer single consumer synchronization in hardware using Full/Empty bits. Coupled with additional state information in the form of extended memory states, the architecture allows all kinds of memory operations to take place. However, the situation gets
complicated when multiple producers or multiple consumers try to synchronize using a single memory location. It therefore becomes necessary to introduce additional software or hardware mechanisms to support such semantics.

A possible hardware mechanism would be to design a smarter memory controller to queue up multiple requests and process them when appropriate replies arrive. An alternative would be to provide similar mechanism using a software handler mechanism. This would primarily act as a synchronization trap handler that will be invoked when the hardware realizes it cannot satisfy some requests. A more suitable design would support the common case in hardware and move the rest to software.

The above mechanisms need to be in place when designing a functionally complete and correct system. This would enable users to exploit the strengths of the system. The main focus of this thesis is to exploit the strengths of the system to come up with synchronization libraries that are fine tuned for performance.

1.4 Objective

The primary objective of this thesis is to explore scalable synchronization mechanisms for architectures that support light-weight multithreading capability coupled with a balanced memory subsystem (LWP and PIM-based architectures). We would like to explore this design space for efficient low overhead implementations of mutex and barrier implementations that scale as the number of threads increases. We plan to achieve this with a combination of hardware and software techniques depending on the target requirements.

We expect to show that the LWP architecture lends itself to designing highly scalable synchronization mechanisms and provide significant performance benefits over current implementations. Our experiments will help validate key assumptions, identity bottlenecks and propose alternative mechanisms to help understand the
problem of scalable synchronization in the context of the target architecture. To summarize, we would like to be able to answer some of the following questions:

- How best to implement current synchronization mechanisms in the LWP architecture?
- What sort of additional hardware or software support should be added to the LWP architecture to enhance the implementation?
- How does the LWP implementations compare to that on current architectures/ISAs?
- Are there any new techniques that can be developed for LWP architecture?
- Are any of these LWP based ideas applicable in conventional systems?

1.5 Approach

The following sections describe the steps taken in this thesis to realize the above objectives. We evaluate the proposed hardware and software techniques using a combination of experimentation and simulation techniques. We provide insight into the problem from different viewpoints by exploring the design space with a number of programming models, ISAs and micro-architectures. Further a wide variety of workloads representative of typical scientific problems is used in our experiments to quantify the effectiveness of our strategies.

1.5.1 Scalable synchronization mechanisms for LWP

We evaluate the performance of synchronization mechanisms for two kinds of LWP based system. They are a homogeneous All-LWP system and a LWP based heterogeneous processor system. We compare our implementations against baseline SMP and CMP systems. To perform our analysis we have used the Structural
Simulator Toolkit (SST) and modified its threading library to implement scalable synchronization mutex and barrier mechanisms. A detailed overview of the concept, approach and performance evaluation methodology is available in Chapter 5.

1.5.2 Active Graph data structures

Active Graph data structures exploit key ISA/microarchitectural characteristics of the LWP architecture to effectively increase the amount of concurrency for a number of parallel operations such as neural nets, parallel prefix operations, collective operations etc. In this thesis, we design and evaluate the idea of synchronization mechanisms that use active data structures to speedup these operations and reduce their overheads. We have extensively used the DimC/SALT framework for evaluating the Active Graphs based synchronization techniques. Details on the concept of Active graphs and the synchronization mechanisms based on them can be found in Chapter 6. The chapter also presents some performance results and analysis.

1.5.3 Thread migration to improve cache utilization

Some operating systems in conventional shared memory machines (specifically SMPs) perform thread migration to balance the load among the system’s processors. In this thesis we explore the applicability of thread migration in the context of better synchronization performance and cache utilization. This is loosely based on the idea of “traveling threads” which involves moving the thread to the location of the data rather than fetching the data to the location of the thread. We use synchronization points (specifically addresses of the mutex variables) to determine if we can migrate threads in SMP systems. Chapter 7 presents the concept, implementation and performance results of this study on a four-node Itanium SMP machine running the Linux 2.6.9 kernel.
1.5.4 Benchmark Suite

The benchmark suite that drives all our simulations/experiments consists of a combination of seven hand-coded synchronization microbenchmarks, the SPLASH2 Benchmark suite and the OpenMP version of the NAS parallel benchmarks. All these codes are modified appropriately to be able to compile and run on the target platform/simulation tool (Linux/IA-64, SST, DimC/SALT). The benchmarks are explained in the Chapter 3 and the compiler or platform specific modifications and input parameters are explained in the corresponding chapters.

1.6 Thesis Organization

This chapter gives an introduction to the rest of the work presented in the thesis. The rest of the thesis is organized as follows. Chapter 2 presents some of the previous work done in the area of PIM-based systems multithreading and synchronization protocols. Chapter 3 gives a brief overview of the Light-weight memory system under consideration. Chapter 4 presents scalable synchronization mechanisms for two kinds of LWP based systems. Chapter 5 extends the idea of active graphs to synchronization mechanisms. In Chapter 6 we present an analysis of the applicability of traveling threads to a conventional system running Linux kernel. The last chapter draws some conclusions and lists some possible avenues for future work.
CHAPTER 2

BACKGROUND WORK

This chapter presents some of the background work in the area of PIM architectures, multithreaded architectures, and fine-grain synchronization mechanisms. The chapter is organized as follows: first we present the existing work on PIM architectures. Next we describe multithreaded and multicore processors and give examples of each. The rest of the chapter deals with synchronization mechanisms.

Under synchronization, we first define basic concepts, explain the different synchronization waiting algorithms, define barrier and mutex based synchronization mechanisms. As an example, we also list the synchronization API in POSIX Threads library. Finally we describe some of the scalable barrier and mutex synchronization algorithms which we later use.

2.1 Processor-In-Memory Systems

2.1.1 Rationale behind Processor-In-Memory systems

Ever since its introduction in the 1970s, the DRAM technology has been driven by cost and capacity while the logic technology has been driven by speed, creating a ever widening gap between faster logic and slower memory. According to [13], the performance improvement in latency of commodity DRAM has been less than 7% year. On the other hand, until recently processors have realized a performance improvement of more 80% per year due to both technological and architectural
improvements. If these improvements are equated to clock rate then this implies that memory accesses take more processor clock cycles to satisfy memory requests. This widening performance gap is known as the processor-memory performance gap or more commonly the memory wall [14].

The commercial microprocessor has undergone a number of (micro-)architectural modifications over the years to primarily avoid/tolerate the latency of a memory access. One of the most important approaches has been to provide large on-chip and off-chip SRAM caches to avoid accesses to the memory. Though in recent years fabricating large SRAM caches has been technologically possible, SRAM consumes a lot of power and hence lead to higher thermal dissipation by the processors. Further, caches are useful only when applications have regular temporal and spatial access characteristics to memory. This is not generally the case in most scientific and high performance codes. Finally in a multiprocessor system, a cache coherency protocol is required to keep the data in different caches coherent, which adds additional complexity to the system and increases the latency of remote access.

Next, a number of look-ahead optimizations were added to perform “useful” work instead of the processor being idle during access to the memory. For example, superscalar processors use techniques such as dynamic scheduling, branch prediction and speculative execution leading to additional implementation overheads and under utilization of resources. Further architects have realized that the premise of these techniques i.e. Instruction-Level Parallelism (ILP) doesn’t really seem to exist in both commercial and scientific applications to the degree that these architectures can really exploit them.

On the other hand, the PIM technology overcomes the memory wall problem by reducing the physical distance between the processor and memory. Traditionally, a Processor-In-Memory (PIM) system is based on the concept that the memory
is integrated in the same die as the processor core, providing the processor with higher bandwidth and lower latency to the memory. A number of PIM architectures have been proposed and implemented both in academia and industry. Some of the architectures include, and not limited to, EXECUBE [1], Terasys [15], DIVA [16], VIRAM [17], Gilgamesh MIND [18], CRAM [19], FlexRAM [20], MIP [21].

2.1.2 Technology viewpoint

PIM technology can be classified into two categories based on whether the logic technology or DRAM technology is used as the fabrication process technology. When DRAM-based PIM process is used for fabricating both the logic and the DRAM on a single die the resulting design maintains the density of DRAM but the speed of the logic is greatly reduced. In practice, a logic circuit fabricated in a logic process is at least twice as fast as a comparable circuit fabricated in a high-yielding DRAM process. However when DRAM is design is optimized for density, DRAMs have a factor of 6 to 20 or more capacity per unit area when compared to SRAMs. Therefore technology is most suited for applications that are memory intensive and corresponding architectures that provide rich memory semantics more than just stock load and store instructions.

On the other hand the advantage of Logic-based PIM technology is that the speed of the logic circuit is not sacrificed though the density of the memory integrated into the same die may be affected. The Logic based PIM process can also be classified based on whether the on-chip memory is SRAM or DRAM. The manufacturing concepts of the former is well understood because of its pervasive use for caches. In general, plain SRAM macros do differ from caches primarily due to the addressing schemes used. Though SRAM memory is as fast as the corresponding logic circuits, the primary problem with SRAMs is that it is not as dense as DRAMs. The density
advantage of DRAM provides from four to eight times larger capacity when the logic based process is used.

A recent paper [22] makes a case for Logic-based eDRAM and compares it against using both SRAM memory or using the DRAM process technology. The paper argues that in future as feature size shrinks the performance gap between SRAM and eDRAM would reduce. Further, DRAM has additional benefits of power over SRAM since it requires lesser number of transistors to build a DRAM cell compared to a SRAM cell. Other technologies such as ZRAM [23] have also provided some promising results in combining memory and processing cores. However it will be interesting to see which of these different technologies becomes the most suited for a wide range of applications and allows low cost high density implementations possible.

The problems and solutions presented in this thesis are for the most part independent on whether DRAM-based PIM process is used or a Logic-based eDRAM process is used. However we recommend DRAM-based PIM process if low cost high yield technology support is readily available. The reasoning is as follows: we are targeting memory intensive applications where the focus is on providing better support for memory based operations. In this regard we expect that the size of the DRAM will play a more important role than the speed of the logic and hence the choice of DRAM-based PIM process over Logic-based eDRAM process.

2.1.3 Examples of next generation PIM Systems

PIM-based architectures have an exciting future ahead finding applications within commercial DRAMs and high-performance supercomputers. The Bluegene/L supercomputer occupies five of the fastest ten machines in the Top500 list of supercomputers [24]. The primary processor consists of two PowerPC 440 processor
each with a 4MB EDRAM cache. The original proposal of the Cascade System (funded by DARPA HPCS program) included the specification for a Light-weight processing subsystem based on PIM technology. Another new architecture called the Cyclops [25] is being developed by IBM T. J Watson Research center for cellular computing applications. The main component, called Cyclops chip, supports multiple threads in multiple cores and the on-chip memory (8MB) is shared among the different cores.

The next generation DRAM interface from Rambus, called XDR2, uses a technique called DRAM microthreading to effectively use the data bus bandwidth by additional control logic on the DRAM. With DRAM microthreading, the XDR2 interface can service four different read requests for four different locations in memory in one and the same read. So instead of producing a single 32-byte block of contiguous data in response to a request for a single 8-byte block of data, it can produce four 8-byte blocks that have been specifically requested by the processor. The four non-contiguous read requests can come from a single thread or different threads, for example from four concurrently running threads in a dual-core SMT system.

The Xenon Chip [6] that powers the Microsoft Xbox 360 gaming console has a GPU co-processor designed by ATI. It is interesting to note that a relatively large amount of EDRAM (10MB) resides on a separate die from the main GPU logic. The EDRAM chip also contains some dedicated alpha blend, z-test and antialiasing logic. The EDRAM die contains 100 million transistors in an NEC 90-nm process. The main GPU die contains 232 million transistors in TSMC 90-nm gate technology. Both the GPU die and the EDRAM die are integrated with the MCM technology.
2.2 Multithreaded and Multicore Architectures

This sections gives a brief overview of existing multithreaded and multicore architectures. We define the basic terms and give examples for each type. More details can be found in the references cited below.

2.2.1 Multithreaded Architectures

Techniques that aim at attacking the memory latency problem either attempt to reduce the latency of memory access or hide/tolerate the latency. Caching is a classical example of a latency reduction technique. Caching attempts to reduce the latency by making the data available from a smaller and faster storage closer to the processor. However, latency hiding techniques attempt to fill the delay slots with other useful work. Techniques such as prefetching and multithreading fall in this category. Of special interest to us in this thesis is multithreading.

A multithreaded architecture can execute instructions from different threads of control within a single pipeline. A thread is primarily a stream of instructions that have their own program counter, a set of working registers and share an address space with other threads. Multithreaded architectures primarily tolerate long latencies by switching threads automatically based on some hardware thread switching policy. Not surprisingly multithreading can also improve utilization by masking pipeline dependencies, cache misses, network latencies, and filling unused instruction issue slots to name a few.

Classical multithreading mechanisms can be classified into Fine-grain (or interleaved) multithreading and Blocked multithreading based on when the processor chooses to switch threads. Fine-grained multithreaded processors issue instructions from different threads each cycle. Examples of such machines include HEP [26], Cray MTA [27], Cray Eldorado [2] and MIT M-Machine [28]. On the other hand,
blocked multithreaded usually switch threads only when a long latency event occurs. For example, the MIT Alewife’s SPARCLE processor [9] support four thread contexts in hardware and switches between them on a remote cache miss or a failed synchronization event.

Another category that is popular nowadays is the *Simultaneous Multithreading (SMT)* architectures [29]. Unlike the above two categories, SMT architectures issue from more than one thread each cycle. By combining multithreading with wide-issue superscalar processing, SMT designs targets the under utilization of superscalar processor by utilizing a larger part of the issues bandwidth. The IBM Power5 [4] is a good example of a two-way SMT machine.

All the categories of multithreading discussed above fall under *Explicit multithreading*. These rely on the programmer identify threads as part of their code. This is usually achieved using standard APIs such as POSIX Threads (PThreads) Library. The other category, called *Implicit multithreaded architectures*, which try to extract threads from sequential programs applying techniques such as Threadlets [30], and thread level speculation [31]. The primary purpose of these techniques is to boost the performance of sequential programs. For the purposes of this thesis, we primarily deal with explicit multithreaded architectures that support fine grain threads. Readers can refer to [32] for a more complete survey of ideas, classifications and examples of some explicit multithreaded architectures.

2.2.2 Multicore Architectures

*Multicore* chips (a.k.a Chip Multi-Processors or CMPs) uses advances in CMOS technology to integrate multiple processing cores on a single die. The development of multicore architectures have been driven by technical limitations of single core processors. Improvements in CMOS technology, as predicted by Moore’s Law, has
enabled chip designers to integrate more transistors per unit chip area. However, as previously mentioned two issues have dampened these improvements.

First, the power dissipation of processors have become exceedingly high. This includes both dynamic power as well as leakage power. Second, the diminishing returns of making designs more complex has made architects and chip designers rethink the way architectures are implemented. The multicore approach currently seems to be the direction pursued by both academia and industry to tackle these problems. The general idea is to utilize the increasing chip resources more efficiently by replicating multiple simple cores on a single die. Multicore designs usually have upwards of two cores on a single die and they usually share a large L2 cache. Each of the cores may be clocked at a lower rate than most contemporary single core processors. This is primarily for reducing the power dissipation of the entire chip.

The idea of multicore chips is not new. EXECUBE [1] is arguably one of the first design to integrate “complete” processing cores on a single chip and also uses DRAM process technology to achieve that goal. Currently a number of designs such as Cray Eldorado [2], Sun Niagara [3], IBM Power5 [4], Intel Core Duo [33] and AMD [34] have successfully integrated two to eight cores in a single chip. Not surprisingly the next generation gaming consoles from Sony[5] and Microsoft[6] use multicore chips. Other designs such as systolic architectures [35] and few Digital Signal Processing (DSP) processors have also integrated multiple “cores” on a single chip.

Multicore designs can be classified into Homogeneous and Heterogeneous multicore processors. This classification is based on the number and type of cores that make a multicore chip. Homogeneous designs are ones where all the cores are the same. The Microsoft-IBM Xenon processor [6] is a homogeneous design consisting of three Power5-based cores sharing a 1MB L2 cache. Most desktop multicore ini-
tiatives from Intel, AMD and IBM currently pursue this path for their near-term future designs.

On the other hand, heterogeneous designs have two or more different types of cores and depending on the target environment each type may be replicated different number of times. System-On-a-Chip (SOC) designs that have been popular with the embedded systems community can be classified as heterogeneous multicore designs. The Sony-IBM Cell processor [5][36] is a good example of a heterogeneous design. It consists a one 64-bit POWER4-like core with limited floating point performance coupled with eight smaller SIMD-based cores (called Synergistic Processing Elements (SPE)).

The idea of multicore designs is currently not without problems. The software tools and programming languages for utilizing such designs are way behind compared to the hardware development. This has hampered the quick adaptation of these processors even though they provide significant performance and power benefits. It should be noted that single threaded programs may actually perform worse than on single core processors because of lower clock rate and less aggressive designs. However since different applications can be executed on different cores, the overall performance of the system may be better.

2.3 Synchronization Mechanisms

On shared memory architectures (and programming models) processes/threads communicate by sharing data structures. Consistency of these data structures is ensured by using synchronization constructs that protect the data from conflicting operations. Parallel programs at the minimum require two types of synchronization techniques [7]:

23
**Mutual Exclusion** ensures that only one thread or process has access to perform certain operations on a given shared object. This enforces a serialization of execution on the participating processes/threads. Locks (or mutexes) are the most commonly used mutual exclusion constructs.

**Events** is used to signal a set of threads that some point of execution has been reached and that they can proceed with their execution. Synchronization barriers, condition variables, and flags are the most common examples of event synchronization constructs.

Locks and barriers are the most popular programming constructs used to coordinate threads in any parallel/multi-threaded system. These constructs have a wide set of implementations in parallelizing languages and compilers (OpenMP), in thread libraries (Pthreads), in system libraries (PARMACS, Linux futex) and in hardware (Cray T3E, Cyclops).

*Locks* serialize critical execution and provide atomic access to shared data by guaranteeing mutual exclusion. Lock based synchronization is one of the most primitive synchronization constructs and is used to build more complex ones such as monitors.

A *barrier synchronization* construct ensures that all processes/threads have arrived at a specific point in program execution before allowing any of the participating threads to proceed with the next phase of the computation. Though not scalable, the simplest and the most common barrier implementations are achieved using locks protecting a shared counter/flag. Additionally thread join operations can be semantically considered to represent a barrier. A thread join (as in Pthreads **pthread_join**) is primarily used by a parent thread to wait for *all* its child threads to complete its work.
Scalable synchronization algorithms primarily ensure that the cost of synchronization increases logarithmically or at most linearly with the linear increase in number of participating threads or processors. The current design space for synchronization mechanisms consists of efficient hardware mechanisms that are used to build flexible software mechanisms. In addition to the basic algorithms, synchronization mechanisms must also deal with issues such as total number of network and memory traffic generated, fairness, number of operations in the critical path, etc. to name a few.

2.3.1 Waiting algorithm

A waiting algorithm determines the action taken by a thread when it is waiting for a synchronization operation to complete [37]. Synchronization constructs can be divided into blocking constructs and busy-wait (or spinning) constructs based on the waiting algorithms. Blocking constructs de-schedule waiting threads until the operation completes, while busy-wait constructs allows the thread to execute repeatedly in a loop to test the completion of the operation.

Blocking is usually performed by the scheduler at the library or the OS level, compared to busy-wait techniques usually implemented in hardware. This is because of the complexity involved in implementing signaling mechanisms and the cost of saving/restoring a thread’s state in hardware. Threads that block are added to a software queue associated with pending synchronization operations.

Traditionally, spin-waiting techniques have provided better performance than blocking techniques primarily because of the overheads involved in performing the latter in software. In this regard, most shared memory architectures implement the basic mutex and barrier synchronization mechanisms using some sort of busy-wait algorithms. The instructions such as Load-Locked (LL) or Store-conditional (SC)
are available in most architectures to implement spin-wait techniques. For example, the PowerPC ISA provides two instructions, namely `lwarx` and `stwcx`, that can be used in a loop to implement primitive synchronization operations.

In multithread processors, the basic waiting algorithms can be extend to provide support for additional mechanisms such as switch-spinning and switch-blocking. *Switch-spinning* allows the processor to rapidly switch to another processor resident thread in a round-robin fashion. On the other hand, *switch-blocking* disables the thread’s context and switches to another processor resident thread. Since the thread’s context is not moved, this mechanism incurs a fixed cost. The MIT Alewife [9] machine provides support for these techniques in addition to conventional spinning and blocking techniques.

*Two-phase* waiting algorithms [38] take a hybrid approach by splitting the waiting time between a poll phase and a block phase. The algorithm spins for a duration of time and then blocks if it does not receive a reply within that time period. The actual time spent on each phase can accurately determined only during runtime. However the cost of determining when to switch between phases incurs a high runtime overhead. For this reason, static two-phase algorithms are more popular than adaptive or runtime algorithms. The Cray MTA [27] also employs a two-phase waiting algorithm and uses a trap handler to queue/wake-up threads for pending synchronization operations.

2.3.2 Example: PThreads synchronization API

The POSIX Threads (PThreads) Library is a standardized API for writing multithreaded programs in C language. It consists of a set of C language programming types and procedure calls. The Pthreads API provides support for three main synchronization constructs, which are:
The **pthread_mutex** is used to implement mutual exclusion (mutex) or a “lock” mechanism. This is to restrict accesses to shared data objects protected by a mutex to just one thread that holds the mutex.

**pthread_barrier** is used to implement barrier synchronization.

**pthread_cond** allows threads to synchronize based upon the actual value of data instead of controlling access to the data using mutexes.

In addition to these three constructs, additional support is also provided for Read-Write Locks and Try Locks. Table 2.3.2 summarizes the different types and function calls for some of the important synchronization constructs. Though the specification defines barrier synchronization calls, most implementations of Pthreads do not provide support for the **pthread_barrier**.* calls. Usually, barriers are built from a combination of **pthread_mutex**.* and **pthread_cond**.* calls.

### TABLE 2.1

**Pthreads Synchronization API**

<table>
<thead>
<tr>
<th>PThread type/call</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pthread_mutex.t mutex</strong></td>
<td>Declare a mutex variable <strong>mutex</strong></td>
</tr>
<tr>
<td><strong>pthread_mutex_init(mutex, attr)</strong></td>
<td>Initialize <strong>mutex</strong> variable and set its attributes</td>
</tr>
<tr>
<td><strong>pthread_mutex_lock(mutex)</strong></td>
<td>Acquire a lock on the specified <strong>mutex</strong> variable</td>
</tr>
<tr>
<td><strong>pthread_mutex_unlock(mutex)</strong></td>
<td>Unlock the <strong>mutex</strong></td>
</tr>
<tr>
<td><strong>pthread_cond_t cvar</strong></td>
<td>Declare a condition variable <strong>cvar</strong></td>
</tr>
<tr>
<td><strong>pthread_cond_init(cvar, attr)</strong></td>
<td>Initialize <strong>cvar</strong> and set its attributes</td>
</tr>
<tr>
<td><strong>pthread_cond_wait(cvar, mutex)</strong></td>
<td>Wait until <strong>cvar</strong> is signaled</td>
</tr>
<tr>
<td><strong>pthread_cond_signal(cvar)</strong></td>
<td>Signal another thread waiting on <strong>cvar</strong></td>
</tr>
<tr>
<td><strong>pthread_cond_broadcast(cvar)</strong></td>
<td>Signal all threads waiting on <strong>cvar</strong></td>
</tr>
<tr>
<td><strong>pthread_barrier.t barrier</strong></td>
<td>Declare a barrier variable <strong>barrier</strong></td>
</tr>
<tr>
<td><strong>pthread_barrier_init(barrier, N)</strong></td>
<td>Initialize <strong>barrier</strong> for <strong>N</strong> threads</td>
</tr>
<tr>
<td><strong>pthread_barrier_wait(barrier)</strong></td>
<td>Wait for all <strong>N</strong> threads to achieve <strong>barrier</strong></td>
</tr>
</tbody>
</table>
2.3.3 Example: OpenMP synchronization

The OpenMP programming model [39] is a high performance shared memory programming model supported by a wide range of systems. The OpenMP model exploits loop level parallelism where it divides the parallel program into a “master” thread and multiple “slave” threads. The primary purpose of the “master” thread is to divide the available loop level parallelism among “slave” child threads using a variety of static and dynamic scheduling policies. Table 2.3.3 lists some of the OpenMP synchronization directives.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATOMIC</td>
<td>Specifies a memory location must be updated atomically</td>
</tr>
<tr>
<td>CRITICAL</td>
<td>Specifies a region of code as a critical section</td>
</tr>
<tr>
<td>BARRIER</td>
<td>Specifies a barrier synchronization point</td>
</tr>
</tbody>
</table>

In addition to this OpenMP automatically inserts a barrier at the end of sections of code that has to be executed in parallel. These parallel sections can be specified using the PARALLEL directive or work sharing constructs (DO/for, SECTION, SINGLE) or a combination of both (PARALLEL DO/for, PARALLEL SECTION). Only the master thread is allowed to execute beyond that point.

2.4 Barrier Synchronization Algorithms

The support for efficient and scalable barriers is not new and researchers have been investigating barriers in both hardware and software for a long time now. This
section provides a brief overview of some of the barrier algorithms in literature. Our
description/code is based on the optimized versions of the algorithms as presented
in [40] and [41].

2.4.1 Hardware barrier synchronization

Barrier synchronization is a high latency operation and a number of machines
have proposed or implemented fast barrier mechanisms in hardware. A number of
systems have implemented either dedicated barrier networks [42][43] or provided
hardware support within existing data networks [44][45].

More recently, the Cyclops chip [25] supports a hardware barrier mechanism
implemented using special purpose registers that all threads have access to. Experiments
on the Cyclops systems has shown that fast barrier hardware can improve
performance of SPLASH2 by 10%. Though hardware barrier synchronization
provides a performance benefit, the cost of implementation make it prohibitive for a
more wider acceptance. Additionally, hardware barriers may be inflexible when
the number of threads participating in the barrier keeps changing or when different
threads call different sets of barriers.

2.4.2 Centralized barriers

The simplest and the most popular implementations of a barrier construct is
a centralized barrier. The centralized barrier can be implemented using atomic
instructions or using locks. The following code shows a centralized barrier im-
plemented using atomic fetch_and_decrement operation. Here, NUM_THREADS
defines the number of threads participating in the barrier.

```c
typedef unsigned int bool;
unsigned long counter = NUM_THREADS;
bool sense = TRUE;
```
central_barrier() {
    static bool local_sense = FALSE;
    if ( fetch_and_decrement( &counter ) == 1 ) {
        counter = NUM_THREADS;
        sense = local_sense;
        //toggle global flag for next iteration
    } else
    while ( sense != local_sense );
    local_sense = ~ local_sense;
    //toggle local flag for next iteration
}

The role played by the sense flag is an important one and is worth mentioning here. In a many-phase algorithm, it is very likely that the same barrier is used repeatedly. In such a situation, each barrier instance begins and ends with the same values for the shared variables. The sense flag ensures that consecutive barriers can proceed without any overhead of setting up the values by reversing the waiting condition for the threads. Without it each processor has to spin twice every barrier instance; once to ensure that all threads have left the previous barrier and again to ensure that all threads have arrived at the current barrier.

2.4.3 Combining tree barriers

The combining tree barrier uses two trees: the arrival tree and the wake-up tree to synchronize threads in a scalable manner. Writes into the arrival tree are used to determine if all processors have reached the barrier, and reads out of the wake-up are used to allow the threads to continue with the next phase of the computation. Threads are divided into groups and each thread group is assigned to a unique leaf of the tree. The number of threads in each thread group determines the fan-in of the tree and is usually four.

Each thread starts at the leaf of the arrival tree and increments a shared counter at that node. The last thread to arrive at a node continues up to the next level.
Figure 2.1. Four threads in a Dissemination barrier

The thread that arrives at the root of the tree begins the wake-up process. Each thread retraces its path downward in the tree waking up its siblings at each level until it reaches the leaf. The authors of [40] point out a critical shortcoming of this approach, namely the requirement for threads to spin on memory locations that cannot be statically defined and on which other threads also spin.

2.4.4 Dissemination barriers

The butterfly barrier [46] is based on the FFT butterfly structure. Each thread in this barrier participates in pairwise synchronizations with log of the number of threads participating in the barrier. When the number of threads is not a power of two then threads synchronize twice as much.

The dissemination barrier [47] improves on the butterfly barrier. It employs a more efficient synchronization pattern since the threads no longer perform pairwise synchronizations. In round $k$, thread $i$ signals thread $(i+2^k) \mod \text{NUM_THREADS}$. It then waits for another thread $j$ that should be signaling this thread before proceeding to the next round. Thus, regardless of the number of threads, each thread participates only in log of the total number of participating threads.
Figure 2.2. Eight threads in a Tournament barrier

Figure 2.1 shows the two levels in a dissemination barrier for synchronizing four threads. The results of [40] show that dissemination barrier to be the most suitable for CC-NUMA machines without broadcast.

2.4.5 Tournament barrier

Tournament barriers [47] are similar to a combining tree barrier of fan-in two in that threads start at the leaf nodes of a binary tree. However at each stage the winning thread that should continue up the tree is statically determined and is not necessarily the last thread that arrives at any node. In round \( k \) thread \( i \) sets a flag awaited by thread \( j \) where \( i = 2^k (mod 2^{k+1}) \) and \( j = i - 2^k \). Thread \( i \) drops out of the tournament while thread \( j \) continues to the next round one level up the tree. The optimized version of the tournament barrier uses a binary wake-up tree such that each thread spins on its own set of contiguous statically allocated flags.

Figure 2.2 shows an example of a tournament barrier for eight threads. The straight lines represents the winner, the dashed lines represents the loser and the
dotted lines represents the wake-up signal for the thread that lost the tournament. For example at node $C$, thread 0 and thread 1 compete. Thread 0 wins the tournament and moves up to node $E$ where it competes with thread 2. During the wake-up phase, thread 0 retraces its path and wakes up thread 2 at $E$ and then thread 1 at $C$. Threads continue the tournament process until one thread reaches the root.

2.4.6 Tree barrier

The tree barrier [40] uses two trees i.e. the arrival and the wake-up trees as in the previous algorithms. However unlike the previous barriers, each node in the tree is now assigned to a thread. Upon arrival at a barrier, each thread waits for its child threads to arrive at their respective child nodes. Once all of a nodes children have arrived the nodes associated threads sets the appropriate flags in the nodes parent.

When the thread at the root node notices all its children have arrived it starts the wake-up process by setting appropriate flags in each of its child nodes waking up the corresponding child threads. At each level newly released threads release all of their child threads before leaving the barrier, and this process continues down to the leaf nodes of the tree. The tree based barrier also ensures that the threads spin
on locally accessible flag variables, perform theoretical minimum number of network transactions on machines without broadcast and require space linear to the number of threads that need to be synchronized.

Figure 2.3 shows the construction of a tree barrier for eight threads. Though the figure depicts a fanin of two for each node, previous studies have shown better performance for fanin of four. The numbers in each node represent a unique thread id. The straight lines represents the signals that indicate a thread’s arrival at the barrier and the dashed lines presents the wakeup process. Threads are ready to continue with their computation once they have woken up their immediate children.

2.4.7 Adaptive and Fuzzy approaches

The tree based barriers presented above (tree barrier, tournament barrier, dissemination barrier and combining tree barrier) have shown to provide a significant performance improvement over the centralized barriers since they distribute the contention and process thread arrivals in parallel. These barriers are also called as logarithmic barriers since they usually require $O(\log NUM\_THREADES)$ operations on their critical path.

Centralized barriers have an advantage over the logarithmic barriers when it comes to time taken by the last thread to signal the achievement of the barrier. Usually, the last arriving thread in a centralized barrier requires only constant time to do this. However, logarithmic barrier algorithms incur a $O(\log NUM\_THREADES)$ delay even after the last thread has arrived. For example, in tree-based barrier, the threads that are assigned to the leaf nodes do not discover that the barrier has been achieved until the threads higher in the tree signal to them.

Adaptive combining tree [48]and fuzzy barriers [49] have been proposed to exploit more concurrency during the arrival of threads at a barrier synchronization point.
In adaptive combining tree barriers, threads that arrive early at the barrier modify the tree so that later arrivals can start closer to the root of the tree. If threads are skewed sufficiently, the last thread that arrives at the barrier require almost constant time to signal other threads that the barrier has been reached.

Fuzzy barriers extend this by dividing the arrival process at a barrier into two phases. In the first phase, threads guarantee they have completed all operations on which other threads depend on. Threads that have completed the first phase then wait for other threads to complete the operations that they depend on. A modified version of these barriers that reduce the number of operations per node and the memory and interconnect traffic is presented in [41].

The key to ensuring correctness in these barriers is to guarantee that two threads do not modify the same part of the tree. For this purposes, these algorithms use a range of techniques from locking individual tree nodes to using atomic updates using primitives like fetch-and-store. The fetch-and-store primitive performs an asynchronous, wait-free update by atomically swapping the contents of a location with a value and returning the original value in that memory location.

2.5 Scalable Mutex Synchronization algorithms

This section provides a brief overview of some of the hardware and software locking mechanisms in literature.

2.5.1 Test-and-Set and Test-and-Test-and-Set locks

One of the most common implementations of a lock consists of a loop that checks a memory location to determine if a lock is held. Each processor repeatedly tests a memory address that acts as a lock and sets it to indicate it is held if it finds the lock is free. This is the simplest lock and was originally implemented using the test-and-set instruction. This instruction was first available in IBM System/360
[50] and hence the name Test and Set (TS) lock. The TS lock is efficient when the lock is free but has a number of disadvantages when the lock is contended by other processors. It does not guarantee fairness and generates a lot of unnecessary coherency traffic even when the lock is held.

Exponential Backoff is a delay policy often applied to TS lock to reduce the number of requests to a contended lock. According to this policy, after each failure, processors wait for successively longer periods of time before issuing another request to the lock. The problem with this approach is that processors may wait longer than necessary and hence increase the overall time required to perform the synchronization operation.

A variation of the TS lock is the Test and Test and Set (TTS) lock. The TTS lock performs a read of the lock before attempting to acquire the lock. The TTS lock allows processors to spin on local read-only copies until the processor that holds the lock releases it. When the holding processor releases the lock, the other processors lose their read-only copy of the lock. Eventually, one of the processors succeeds in the TS operation when its first test operation succeeds.

More recently, the availability of LL-SC instructions in PowerPC and Alpha processors have replaced the need for test-and-set instructions and allows more concurrency to be exploited. These instructions use reservation stations to determine if the code between the LL and the SC instruction executed atomically. The LL and SC instructions have to be paired to ensure correct execution.

An LL instruction, reads a memory location and set a reservation bit corresponding to the memory location. For the SC instruction to succeed, it first ensures the reservation its predecessor LL instruction set is still valid. If not it implies some other processor has attempted the LL-SC pair. If the reservation is still valid then the SC succeeds and it saves a value into the memory address. The SC releases the
reservation once it succeeds. The PowerPC ISA uses lwarx and stwcx instructions to implement the LL and SC operations respectively. The GCC inline assembly code for the TTS lock using these PowerPC instructions is shown below.

/* Test and Test and Set Mutex Lock*/

_INLINE_ void LLSC_TTS_mutex_lock(unsigned int *a) {
    asm volatile ("mr r3, \%0\n"
                  "li r4, 1\n"
                  "lwarx r5, 0, r3\n"
                  "cmpwi r5, 0\n"
                  "bne loop1\n"
                  "lwarx r5, 0, r3\n"
                  "cmpwi r5, 0\n"
                  "bne loop1\n"
                  "stwcx. r4, 0, r3\n"
                  "bne loop1\n"
                  "isync\n"
                     : : "r" (a)
                     : "r3", "r4", "r5";)
}

/*Mutex Unlock*/

_INLINE_ void LLSC_mutex_unlock(unsigned int *a) {
    asm volatile ("sync\n"
                  "mr r3, \%0\n"
                  "li r4, 0\n"
                  "stw r4, 0(r3)\n"

37
\ \ release lock

:  "r" (a)
  :  "r3", "r4";
}

2.5.2 Ticket lock

The ticket lock uses two counters to determine if the lock is held or free. The first counter (next_ticket) keeps track of the requests that have already been made for the lock and the second counter (now_serving) keeps track of the number of times the lock has already been released. The processor increments the first counter to determine its “ticket”. Then it spins until the second counter is equal to its ticket. The processor releases the lock by incrementing the second counter which enables the next processor in line to acquire the lock.

Since processors acquire the lock in FIFO order, an exponential back-off may be too excessive. Instead, the ticket lock uses a delay policy called the Proportional Backoff. According to this policy, processors delay for an approximate amount of time proportional to the number of processors already waiting for the lock. The following code segment shows the ticket lock implemented using a fetch-and-add atomic primitive. We use an implementation of the fetch-and-add primitive based on the LL-SC instructions and it takes two arguments: a address and a value. It atomically increments the address by the value and returns the new value.

unsigned int LLSC_fetchAdd(unsigned int *a, unsigned int v) {
  int result;
  asm volatile (  
    "mr r3, %1\n"
      \ \ r3 holds the address a
    "mr r4, %2\n"
      \ \ r4 holds the increment v
    "loop: lwarx r5, 0, r3\n"
    "add r6, r4, r5\n"
      \ \ increment a by v

void LLSC_ticket_mutex_lock(scale_mutex_t *lock) {
    unsigned int my_ticket = LLSC_fetchAdd(&lock->next_ticket, 1);
    while(1) {
        if (lock->now_serving == my_ticket) return;
        sleep(my_ticket-lock->now_serving);
    }
}

void LLSC_ticket_mutex_unlock(scale_mutex_t *lock) {
    LLSC_fetchAdd(&lock->now_serving, 1);
}

2.5.3 QOLB Hardware queue Lock

Queue On Lock Bit (QOLB) [51] is a distributed, queue-based, non-blocking hardware locking scheme. QOLB uses a hardware queue to maintain a list of waiting processors. Pointers to the adjacent queue entries are held in the processor’s cache. Each processor spins on local copies of the lock address and when a lock is released it is directly sent to the processor on the head of the queue. Experiments have shown QOLB to outperform TS, TTS, MCS and CLH locks by around 40%. QOLB requires significant hardware support in terms of non-blocking synchronization instructions, direct node-to-node transfer of the lock, storage of the queue state information and finally the capability of having multiple processors perform operations on the same address without invoking the coherence protocol. Lesser aggressive implementations of QOLB have shown to be better than the software alternatives.
2.5.4 MCS, CLH: Software Queue based algorithms

A number of software queue based locking mechanisms such as Anderson’s Lock [52], CLH Lock [53][54], MCS Lock [40] have also been proposed. All these mechanisms build software queues of waiting processors and guarantee a FIFO policy for transferring the lock between contending processors. These use atomic instructions to determine an unique location on which each processor can spin and hence reduce the traffic on the network.

The MCS lock uses local spinning and uses memory space linear to the number of threads. The CLH lock is similar to the MCS lock and outperforms the MCS lock for large number of threads since it requires one less remote access to release the lock. Pointers are maintained to successor or predecessor nodes and atomic operations such as fetch-and-store are used to insert and delete entries from the queue.

2.6 Evaluation of synchronization primitives

In many applications, synchronization overheads can be hidden since the actual latency of these operations gets overlapped with other useful computation. Usually the time taken to achieve a synchronization point is measured as the time between when the actual request is initiated to when the calling threads receives an appropriate reply.

Figure 2.4 dissects a barrier synchronization point. There are three main time periods represented by the labels A, B and C. The period A denotes the wait time of a barrier algorithm and is the difference between the last arrival and the first arrival. The second period denoted by B signifies the time taken by the last arrival to make the assertion that the barrier has been achieved. Finally, the time period C denotes the wake time or the time taken to wakeup all the other threads waiting
Figure 2.4. Dissecting a barrier synchronization point

on the barrier. Of the three periods, A is a function of the application and the type of barrier algorithm used. The other two periods are just functions of the barrier algorithm.

In general, metrics such as number of operations in the critical path, the number of remote memory operations, memory requirements etc. are also indicative of the performance of synchronization protocols. Specifically, if the lock data structure can be fitted with the contents of a memory location (co-location) helps in better performance. Further locks need to evaluated how well they guarantee fairness. Queue-based locks usually guarantee better fairness than test-and-set locks since they enforce FIFO ordering.
CHAPTER 3

APPLICATION BENCHMARK SUITE

In this chapter we give a brief overview of the various C and FORTRAN benchmarks used for evaluating the overheads of synchronization. The rest of the chapter is organized as follows: first we explain the Pthreads synchronization microbenchmarks to test the scalable mechanisms. Next, we give a brief overview of SPLASH2 benchmark suite which consists of thirteen benchmarks written for evaluating shared memory architectures. Finally, we conclude the chapter with a description of NAS OpenMP benchmarks which uses OpenMP to exploit loop level parallelism.

3.1 Microbenchmarks

We developed seven microbenchmarks that specifically test synchronization mechanisms. These microbenchmarks differ on a wide range of characteristics including the number of locks, the access patterns of shared data structures with the critical section, the complexity of the critical section code and finally optimizations that try to limit false sharing. The first four microbenchmarks all use one lock to protect the critical section and the last three involve multiple locks. All the threads execute the critical section in a loop. Further, all the microbenchmarks are padded appropriately to take care of false sharing.

All the microbenchmarks were designed using the methodology similar to [55] and [56]. Four of the microbenchmarks (Single counter, Multiple counter, Doubly
linked list, Producer Consumer) were previously mentioned in [57] and [55]. We explain each of these microbenchmarks briefly.

**Barrier (barrier)** consists of a single barrier used to synchronize threads across loop iterations. Threads perform some random work between loops to ensure fairness.

**Single Counter(single ctr)** consists of a counter (fits in a cache line) protected by a single lock and all threads increment this counter in a loop.

**Multiple Counter(multiple_ctr)** consists of an array of counters protected by a single lock and each thread increments a different counter (fits in a cache line) in the array.

**Doubly Linked list(doubly_list)** consists of a doubly linked list where threads dequeue elements from the head and enqueues them on to the tail of the list. The enqueue and dequeue operations are performed independent of each other with separate lock acquire and release operations. The doubly linked list is protected by a single lock.

**Producer Consumer(prod_cons)** consists of a shared FIFO (bounded) array protected by a single lock that is initially empty. Half the threads produce items into the FIFO that are consumed by the other half threads. Producers have to wait for free elements in the FIFO whereas consumers have to wait for data to consume before iterating the critical section code.

**Affinity Counter(affinity_ctr)** consists of two locks that protect two counters. During the first phase of each iteration, each of the locks protects one of the counters and during the second phase each of the locks protect the other counter. A barrier is required between the two phase to ensure atomicity.
Multiple Lock Multiple Counters (mlt_lck_mlt_ctr) is similar to the multiple counter microbenchmark, but there are multiple locks protecting different segments of the counter array. The threads dynamically choose the lock to acquire and hence the counter to update depending on its thread id.

Multiple FIFO (multiple_fifo) is similar to producer consumer microbenchmark, but there are multiple locks each protecting a separate FIFO. Each producer/consumer pair is dynamically assigned to a FIFO depending on its thread id. Each thread acquires/releases only the lock corresponding to the queue it is assigned irrespective of whether it is producing or consuming data.

3.2 SPLASH2 Application Suite

SPLASH2 (Stanford Parallel Applications for SHared memory) [58] consists of five kernels and eight applications to evaluate the performance of centralized and distributed shared address space multiprocessors. The SPLASH2 codes utilize the Argonne National Laboratories (ANL) parallel macros (PARMACS) specifying parallelism. For our experiments the macros acted as wrappers around the corresponding POSIX Threads (Pthreads) API. We used the modified versions of [59] and [60] macro implementations to enable us to run SPLASH2 on all our target machines. A brief description of the benchmarks are listed below:

Cholesky is a blocked sparse Cholesky factorization kernel that factors a sparse matrix into the product of a lower triangular matrix and its transpose.

FFT is a kernel representing the complex 1-D version of the radix-$\sqrt{N}$ six step FFT algorithm, optimized to minimize inter-processor communication.

LU-contiguous kernel factors a dense matrix into the product of lower and upper triangular matrices. The factorization uses blocking to exploit temporal
locality on individual sub-matrix elements. Here the matrix is factored as an array of blocks and thus allows blocks to be allocated contiguously.

**LU-noncontiguous** is similar to the LU-contiguous kernel but the matrix is factored as a 2-D array and hence prevents blocks from being allocated contiguously.

**Radix** is an integer radix sort kernel based on an iterative method, performing one iteration for each radix $r$ digit of the keys.

**Barnes** is an application simulating the interaction of a system of bodies (galaxies, for example) in three dimensions over a number of time-steps, using the Barnes-Hut hierarchical N-body method.

**FMM** like Barnes, is an application that simulates a system of bodies over a number of time-steps. It however simulates the interactions in two dimensions using a different hierarchical N-body method called the adaptive Fast Multi-pole Method.

**Ocean-Contiguous** is an application studying large-scale ocean movements based on eddy and boundary currents. It uses a red-black Gauss-Seidel multi-grid equation solver to solve a system of grids represented as 4-D arrays. The grids are implemented to be operated on as 3-D arrays.

**Ocean-Noncontiguous** solves the same problem as Ocean-contiguous but implements the grids to be operated on with two-dimensional arrays.

**Water-nsquared** is an application that evaluates the forces and potentials that occur over time in a system of water molecules. The forces and potentials are computed using an $O(n^2)$ algorithm and uses a predictor-corrector method to integrate the motion of water molecules over time.
**Water-spatial** solves the same problem as Water-nsquared but uses a more efficient algorithm. It uses an $O(n)$ algorithm making it more efficient for large numbers of molecules and imposes a uniform 3-D grid of cells on the problem domain.

**Radiosity** is an application computing the equilibrium distribution of light in a scene using iterative hierarchical diffuse radiosity method.

**Raytrace** is an application that renders a 3-D scene using ray tracing. An image plane of the object is partitioned into contiguous blocks of pixel groups and distributed task queues are used with task stealing. The data access patterns are highly unpredictable in this application.

**Volrend** is an application that renders a 3-D volume using ray casting technique. A cube of volume elements is used to represent the volume and an octree data structure is used to traverse the volume quickly. Data accesses are independent and irregular.

The SPLASH2 documentation also describes a recommended problem size that is a reasonable base problem size that both can be simulated and is not too small in reality for a machine with up to 64 processors. This base data set size is shown in table 3.2 and is used in most of our experiments. The table also lists the number of locks and barriers for a 32-processor machine (reproduced from [58]). Any modification to the base size is explicitly listed as and when necessary.

### 3.3 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) [61] were designed to help evaluate the performance of parallel supercomputers and are derived from Computational Fluid Dynamics (CFD) applications. We used OpenMP implementation of the NAS NPB benchmarks for all our experiments. The latest version (version 3.2) of NAS-OMP
TABLE 3.1

SPLASH2: BASE PROBLEM SIZE AND COMPLEXITY

<table>
<thead>
<tr>
<th>Kernel/Application</th>
<th>Base problem size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cholesky</td>
<td>d750.o input file</td>
</tr>
<tr>
<td>FFT</td>
<td>2^{16} points</td>
</tr>
<tr>
<td>LU (Contiguous)</td>
<td>512x512 matrices</td>
</tr>
<tr>
<td>LU (Non-contiguous)</td>
<td>512X512 matrices</td>
</tr>
<tr>
<td>Radix</td>
<td>262,144 integers, radix 1024</td>
</tr>
<tr>
<td>FMM</td>
<td>16384 particles</td>
</tr>
<tr>
<td>Barnes</td>
<td>16384 bodies</td>
</tr>
<tr>
<td>Ocean-contiguous</td>
<td>258x258 grid</td>
</tr>
<tr>
<td>Ocean-noncontiguous</td>
<td>258x258 grid</td>
</tr>
<tr>
<td>Water-nsquared</td>
<td>9,261 molecules</td>
</tr>
<tr>
<td>Water-spatial</td>
<td>9,261 molecules</td>
</tr>
<tr>
<td>Radiosity</td>
<td>room</td>
</tr>
<tr>
<td>Raytrace</td>
<td>car</td>
</tr>
<tr>
<td>Volrend</td>
<td>head (256x256x128 voxel)</td>
</tr>
</tbody>
</table>

NPB [62] consists of eight FORTRAN codes (UA, BT, SP, LU, LU-HP, FT, CG, MG, and EP) and two C codes (IS and DC). The NPB-OMP is a sample OpenMP implementation based on NPB3.0-SER, the serial version previously known as the Programming Baseline for NPB. The current implementation does not consider nested parallelism. OpenMP directives were added only to the outer-most parallel loops.

A brief summary is listed below:

**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.

**LU-HP** is similar to LU and is a hyperplane based parallelization implementation.

**FT** contains the computational kernel of a 3-D Fast Fourier Transform (FFT)-based spectral method. This kernel performs the essence of many spectral codes and is good for testing long distance communication performance.

**MG** uses a V-cycle MultiGrid method to compute the solution of the 3-D scalar Poisson equation. This problem tests both short and long distance highly structured communication.

**CG** uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication and employs sparse matrix vector multiplications.

**DC** benchmark builds RB-tree to sort tuples from a dataset and is based on a data mining application.

**UA** is the Unstructured Adaptive benchmark involving the solution of a stylized heat transfer problem in a cubic domain, discretized on an adaptively refined, unstructured mesh. This benchmark measures the effect of irregularity in memory accesses by dynamically changing the memory access patterns.

**EP** deals with a large number of Gaussian pseudorandom numbers which is typical in many Monte-Carlo applications. Since it requires almost no communication it falls under the category of “embarrassingly parallel” kernels.
**IS** is an integer sort benchmark and tests a sorting operation that is important in “particle method” codes. This benchmark does not have any floating point arithmetic involved but is used to test both integer computation speeds and communication performance.

The input data sets for each benchmark is available in five different standard sizes: S, W, A, B, C and D. The S is the sample data set and the rest are larger data sets with higher orders of complexity. These input data sets vary from each other based on the size of the data, number of iterations etc. The user can also specify the maximum number of threads that can be used to parallelize the loops. However it should be noted that the actual number of threads activated may less than the user specified number and is determined by the runtime library.
CHAPTER 4

PROPOSED LWP SYSTEM ARCHITECTURE

The first part of this chapter presents a brief description of a proposed LWP system architecture. A *Light-weight Processing Chip* (LPC) consists internally of a large collection of *Light-Weight Processors* (LWPs) and *Embedded Memory Units* (EMUs). These LWPs provide processing sites for the on-chip computational capabilities, while the EMUs provide the storage capabilities.

Each LWP is highly multithreaded, allowing a very large number of threads to be concurrently in execution within both the LWP and the larger LPC. The LWP system is capable of supporting a very large number of threads primarily because of the light weight nature of the threads executing on them. These threads are called Light-Weight Threads (LWTs) signifying that they own very little state and thus are easy and efficient to manage.

The on-chip EMUs provide better opportunities for reducing the latency and increasing the bandwidth to the memory at rates that cannot be realized in a conventional system with separate processor and memory chips. Embedding the memory units on-chip enables satisfying the large number of requests made to the memory by the LWTs at low cost. The system is scalable in the sense that a large number of LPCs can be connected through a scalable interconnection network to enable threads to access remote memory in other LPCs.
The second part of this chapter deals with a simple single thread consistency model for the LWP architecture. In order to reason about the correctness of multi-threaded programs, a specification of the legal ordering of events for memory accesses from multiple threads is required. A Memory Consistency (MC) model provides the contract that the programmer uses for reasoning about the correctness of parallel programs executing in multiprocessors [63]. This specification has two aspects to it: one, the ordering of memory operations within a single thread namely, the single thread consistency model, and the other, ordering of memory operations across multiple threads.

The single thread consistency model, captures the constraints and relaxations imposed by the program order requirement of a single thread. The program order constraint requires memory operations to be issued one-at-a-time and stall the thread for previous accesses to complete before issuing another access. Since this tends to have a negative performance impact, unprocessors (and multiprocessors), have resorted to relaxing some of these constraints by allowing multiple memory operations.

Relaxing the program order requirement has been possible in modern ISAs by architectural features/components such as pipelining, out-of-order execution, caches, write buffers, etc. Additionally, they also require a variety of expensive checks to ensure there are no conflicts whatsoever and to guarantee the order of completion of the operations. Of course, unprocessor data and control dependencies have to be invariably guaranteed for correct program execution.

In unprocessor systems, the programmer and/or the compiler need not be aware of the underlying architectural techniques that enable the relaxations. This is because the last operation that completes is in program order. However, in shared memory multiprocessors, the last operation with respect to relative timing (due to
general interconnect, distributed memory, cache coherency etc.) of events is not clearly defined since there are multiple reads/writes taking place concurrently. For this reason, in explicitly parallel/multi-threaded programs the programmer uses high-level synchronization constructs such as locks, barriers, fences, etc. to enforce the required memory consistency and event ordering constraints across multiple threads.

In the context of a compiler in a explicit multithreaded environment, since the compiler is generating code only for a single thread, it has to be aware of the single thread consistency model for the most part. Nevertheless, it must be ensured that compiler optimizations (for example code propagation, register allocation, etc.) do not violate the ordering constraints of the memory consistency model. The LWP architecture supports a highly multi-threaded environment permitting a large number of threads to execute concurrently. This drives the need for a MC specification since we potentially have an enormous number of different memory operations on the fly within the system at the same time.

The rest of the chapter is organized as follows: first, we present details on the overall LWP system architecture. This is a condensed summary of the actual LWP architecture specified in [10] and [64]. Then we describe the single thread consistency model. This model analysis instruction sequences within a thread. We also list some LWP instructions that can be used for enforcing explicit ordering among different operations.

4.1 Proposed System Architecture

This section presents a brief overview of the various components that constitutes the LWP architecture. First, we present the architecture of the LWP core and the thread model. Next we explain the LWP memory system and also provides details
Figure 4.1. Simplified view of an LWP core

on the extended memory states. Then we have a look at the LPC organization and finally describe the overall system organization.

4.1.1 LWP Core

Each LWP consists of a simple in-order issue pipeline. Figure 4.1 shows a simplified view of an LWP core. The LWP hardware implementation unlike most modern architectures, does not include any of the complex structures such as instruction reorder buffers, branch predictors, write buffers, per-processor counters etc., that
are primarily used to exploit concurrency and reduce latency to the memory. The cost of providing “common” microarchitectural features, across potentially thousands of LWP s and each LWP supporting a large number of LWTs, is probably too expensive.

A *Light-weight Thread* (LWT) is one that has a small set of working registers and runs predominantly on a single LWP core for small to relatively moderate periods of time. A *Thread frame* (or register frame or thread register frame) is the term given to a consecutive unit of storage that contains all the execution state information needed by a single LWT.

The contents of a thread frame governs the thread’s execution and is unique to each thread. The typical size of the thread frame is small, and hence allows threads to be created, copied, suspended or restarted within a very small number of cycles (ideally one). Thread frames also have some special registers to hold instruction pointers, frame pointers etc.

LWTs are radically different from conventional notions of threads in many ways. Some of these differences are primarily possible due to the small state of the thread and the closeness to memory. The actual implementations and uses of these mechanisms can be found in [64] and [10]. We list some of these below:

- LWTs have the ability to migrate to other LPWs on the same LPC or to other LPCs. This is based on the concept of “traveling threads” which allows execution to be placed closer to the data rather than fetching the data to the processing sites.

- The registers in a thread frame are memory mapped. This allows thread-to-thread communication directly through the thread frame. In conventional processors, thread contexts usually is stored in dedicated hardware registers.
• Each register in a thread frame is also enhanced with the full/empty bit. This allows threads to block on its own thread frame or on the frame of other threads. Threads get eventually woken up when they receive an appropriate response.

• There is no explicit hardware enforced ordering on memory operations. The processor, network or memory does not keep track of the order in which memory operations are issued by each thread. The programmer has to explicitly introduce the required ordering.

• Each thread can potentially have a large number of outstanding memory requests. Threads don’t block until they need to use the data that was requested in an earlier memory request.

During its lifetime, a thread may exist in a number of states, which includes free, executing, waiting, blocked and terminated. The waiting state is different from blocked state. In waiting state, threads are ready to issue instructions but do not have sufficient resources to be scheduled at that moment. However, blocked threads are waiting for an external event to complete such as a full/empty bit to be set/reset. Threads in executing or waiting are called as active threads.

Each LWP maintains a queue of active threads in hardware. This queue, called the active queue, represents a list of threads that are ready to issue instructions. At each cycle, the processor issues instructions from one of the threads in the active queue. For performance reasons, each LWP core also has a small amount of storage called the frame cache, usually 8 to 16 entries wide. The purpose of the frame cache is to maintain thread frames of some of the threads in the active queue. Usually the active queue is larger than the size of the frame cache and so spills into memory.
Figure 4.2. Dword: 65-bit long memory word

Based on a user specified thread quanta, threads are swapped between the active queue in memory and frame cache. Irrespective of the actual policy in which threads are allowed to issue instructions it is safe to assume that there exist some sort of a fair scheduling policy for all active threads. This ensures that all active threads make forward progress and there are no starvation issues due to the order in which threads are allowed to issue instructions. Again for performance reasons, each core also has an small SRAM instruction cache (*icache*). The icache is shared among all the threads executing on a LWP core.

4.1.2 LWP Memory system

The fundamental unit of storage for the LWP architecture is a combination of a 64-bit *double word (Dword)* and a 65\(^{th}\)-bit representing the extension bit. Combining the state of the extension bit with different mode fields in the Dword allows us to define a wide range of extended memory states. Figure 4.2 shows the contents of a Dword and the extension bit. The metadata indicates that there is currently no valid value in the memory word but provides information on when and how a valid data may be found.
### TABLE 4.1

**EXTENDED MEMORY STATE DESCRIPTIONS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Memory state</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPR</td>
<td>Blocked pending reply</td>
<td>Indicates a thread is blocked on the Dword. A pending reply will modify the Dword and wake the thread.</td>
</tr>
<tr>
<td>BPD</td>
<td>Blocked pending data</td>
<td>Indicates a thread is blocked on the Dword because it had no valid data. A pending write request will modify the Dword and wake the thread.</td>
</tr>
<tr>
<td>E</td>
<td>Empty</td>
<td>Dword contains the empty value.</td>
</tr>
<tr>
<td>EC</td>
<td>Error code</td>
<td>Dword contains an error code.</td>
</tr>
<tr>
<td>F</td>
<td>Full</td>
<td>Dword contains a valid value.</td>
</tr>
<tr>
<td>FP</td>
<td>Forwarding pointer</td>
<td>Dword contains an address to which all references to this Dword are forwarded to.</td>
</tr>
<tr>
<td>L</td>
<td>Locked</td>
<td>Dword is locked to normal accesses.</td>
</tr>
<tr>
<td>PR</td>
<td>Pending reply</td>
<td>Indicates an operation is pending on the Dword and will result in the modification of its value.</td>
</tr>
<tr>
<td>TOA</td>
<td>Trap-on-access</td>
<td>Access to Dword in this state forks a new thread.</td>
</tr>
<tr>
<td>U</td>
<td>Un-initialized</td>
<td>Dword contains the un-initialized value.</td>
</tr>
</tbody>
</table>
Table 4.1.2 lists some of the extended memory states possible. The LWP Pofo report [64] defines a wider range of extended memory states and their interactions with different memory operations. We shall ignore the other extended memory states here. For the rest of the discussions in this thesis, we consider only two memory states namely the full and empty state. If the state of the memory is full, it implies the Dword contains a value and if it is empty, then it indicates a value is pending for the Dword.

The LWP ISA specification requires every load or store to leave an virtual address behind in the target register, along with an extension bit and mode field that identifies the kind of response that is expected. Further, every load or store instruction must designate a specific register as a response register and set it to “empty”. This is used to capture the programmer-visible acknowledgment from the target memory location and the receipt of such an acknowledgment can be detected when the response register becomes non-empty. Finally, the architectural semantics demand that a LWT must stall itself when it encounters (at the start of the instruction’s execution) a register that is currently marked as empty, and remains stalled until that register receives some sort of response.

For load instructions, the designated target register is used as the response register. All load instructions mark the target register as empty and leave behind the virtual address of the memory operation, until the appropriate response is returned. However, if the target register is already marked as empty at the start of the load’s execution, the thread is suspended before the load is allowed to enter execution, and the target register is set to the empty-blocked thread state. When the thread is restarted, it is restarted at the load, which is then repeated.

For store instructions, the source register may be used as the response register. Since the source register holds the source value, the compiler takes the responsibility
of making an explicit copy when it requires the value to be preserved [64]. As with
the load, if the designated response register of a store instruction is marked as empty
in some way at the time of the store, then the thread is suspended before the store is
allowed to enter execution, and the target register set to the empty-blocked thread
state. When the thread is restarted, it is restarted at the store instruction, which
is then repeated.

An AMO or a Read-Modify-Write (RMW) instruction can be treated as an
atomic access consisting of both a load and a store [65]. Thus, if an AMO returns
a value, it takes the semantics of the load. If it does not return a value, then it
assumes the semantics of the store. We do not discuss AMOs any further in this
chapter, but they can be treated similar to load and store instructions.

The different “flavors” of loads/stores is the outcome of the extended memory
semantics available in the LWP architecture. The load/store instructions listed in
Table 4.1.2 are characterized by their synchronization behavior, which describes the
extended memory state in terms of a precondition and a postcondition state. In
the mnemonics of the instructions listed, the synchronization behavior is specified
as a two-letter suffix, where the first letter is the precondition (state the memory
location must be before the operation is performed) and second is the postcondition
(state the memory location is left after the operation completes). Here, $e$ denotes
empty, $f$ denotes full and $x$ denotes dont care state. A complete listing and formats
of all the LWP instructions is documented in [66].

Loads and store instructions are also characterized by their alignment behavior.
The alignment states whether the data being transferred is a double word (Dword),
single word (word), half word (half) or byte. For the sake of simplicity, we have
assumed the Dword aligned load and store instructions.
TABLE 4.2

DESCRIPTION OF LOAD/STORE INSTRUCTIONS

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld.ff</td>
<td>unsynchronized load double word</td>
</tr>
<tr>
<td>ld.fe</td>
<td>synchronized load double word</td>
</tr>
<tr>
<td>sd.xf</td>
<td>unsynchronized store double word</td>
</tr>
<tr>
<td>sd.ef</td>
<td>synchronized store double word</td>
</tr>
</tbody>
</table>

4.1.3 Thread

Figure 4.3 depicts threads frame blocked waiting for appropriate request/reply to set the full/empty bits. Threads can either block on registers within its own frame, on registers within the frame of another thread or on memory locations. In each case, the full/empty bit is empty and the EMS state is set appropriately set to indicate a blocked thread frame. Threads get woken up when they receive an appropriate parcel.

The “classical LWP” architecture supports single producer and single consumer synchronization in hardware using full/empty bits. When more than one producer or consumer wants to synchronize on a single memory location then the operation becomes too complex to completely support in hardware. The solution for this is to provide some basic support in hardware and allow software to handle the less frequent more complex operations.

A possible strategy is to have the hardware spawn a special handler thread to process the complex operations. The hardware sets a trap on the memory location under consideration so that subsequent accesses may be handled by the software synchronization trap handler. The handler would process requests from a queue and release the trap after all the requests has been successfully handled. From the
software side of things, it is therefore necessary to provide an algorithm that will handle different combination of requests and handle them correctly.

4.1.4 LPC Architecture

Figure 4.4 shows a simplified view of the organization of an LPC chip. Such a chip will have multiple LWPs, EMU macros and other on-chip components such as Parcel Handlers (explained later) etc. EMUs provide the on-chip storage capabilities for reducing the latency and increasing the bandwidth to the memory. Currently, its estimated that the typical on-chip EMUs size may be of the order of 100MB or more per LPC. These macros allow faster access to denser memories, with accesses usually around 10ns range compared to the 80 – 100ns for commodity DRAMs.
Figure 4.4. Simplified view of an LPC

Further, we estimate each internal row to be around 256 bytes wide. This should be sufficient to be able to hold complete thread frames. Additionally, EMUs are similar to DRAM macros in terms of internal organization and also have the concept of open rows. Internally, when a row gets accessed, it is buffered regardless of how many bytes were originally requested. The rows of addresses that have been accessed recently are called open rows. Usually we have one to four rows that can be considered as an open row. Since each access returns only a subset of a memory row, additional accesses to these open rows are much faster. Based on conservative estimates, the bandwidth available from a single memory macro is over 50Gbit/s.

The on-chip EDRAMs may also be backed by additional off-chip DRAM memory. In this case the EDRAM may act as a cache for the off-chip memory. The actual policies that govern these decisions are specific to the implementation and is not discussed here further.
The primary form of communication between LWP's, memory and other system components is through a messaging mechanism called *Parcels*. Figure 4.5 shows the contents of a parcel. Parcels expands the range of memory requests beyond reads and writes to include AMOs and spawning LWTs. The ability to spawn LWTs at remote memory location is particularly useful for thread migration purposes. The contents of a Parcel usually includes a target memory address, a command that has to performed when the parcel gets to the destination, other operands (such as data for a store) and optional payload. Based on the command, an appropriate reply parcel is also generated to be sent to the sender.

Each LPC has at least one *Parcel handler* to generate and handle parcels. The parcel handler may be multiplexed across multiple LWP's. Parcels travel through the system and is forwarded by NIFs, routers and other communication entities until they reach the appropriate destination. This occurs when parcels arrive at the LPC that governs the associated memory address.

The actual mechanisms involved in executing the command is based on the specific implementation. However, the parcel handler inspects the parcel to determine what should be done to execute the command. For simple commands, the parcel handler may be able to take care of it but for operations such as thread spawning, the parcel handler may request one of the LWP's to handle the command execution.

The *Network Interface* (NIF) is a network subsystem that takes care of adding appropriate routing information to out-going parcels and sending them towards their appropriate destination. They play a role in accepting all requests that are destined for the LPCs that's connected directly to it.

A number of mappings is possible for distributing the various EDRAM macros across the different LWP's in an LPC. Figure 4.6 shows three possible mappings between the various LWP's and EMU macros on a LPC chip. For all our experiments
in the following chapters, we use the one-to-one mapping between LWP s and EMU macros. This configuration has the advantage of being closer to a single LWP and hence allows faster access to the memory. However, a potential hotspot on any address within that macro will overload its corresponding LWP. In general, the LWP interconnection network and the LWP-EMU interconnection network may be a bus, crossbar switch or a more complicated on-chip network.

4.1.5 Overall system organization

The primary purpose of the LWP architecture is to build large scale parallel systems. Figure 4.7 shows one possible organization. For consistency of terminology, an LPC chip along with the DRAM it controls is a Section. Now, a Locale is a collection of Heavy Weight Processors (HWP i.e. conventional processors), Sections, NIFs and other chips. The system is scaled by connecting a number of Locales using a high performance network.

A large scale system as the one under consideration will have hierarchical access to remote memory (EDRAM, DRAM) over a scalable interconnect network. From the LWP perspective, the EMU macros it directly controls and the other macros on the same LPC forms the first two levels in this hierarchy. Then comes the DRAM within the same Section. Finally, other Sections within the Locale can be accessed through the Locale interconnection network.

In general, this enables building very large distributed shared memory systems. The current assumption is that there is no global cache coherency protocol. The
Figure 4.6. Possible LWP-EMU mappings
Figure 4.7. Simplified view of an LWP system

actual memory latencies are hidden by the high level of multi-threading supported at the hardware level, integration of LWP and EMUs on the same die and other architectural enhancements such as migrating the thread to the location of the data. This enables supporting a global shared address space system without the need for a cumbersome coherency protocol.

4.2 Single thread consistency model

This section presents a simple single thread consistency model for the LWP architecture and introduces two instructions, namely STE and STAM, for enforcing ordering among memory operations. This serves two purposes: first, for defining a general system-wide MC model and second, to reason about consistency when developing a LWP compiler.
The single thread consistency model primarily provides information on the order of issue/completion of memory access and how that order affects the execution of the thread. Such a model may be unnecessary for architectures that do not allow memory operations to be issued or completed out-of-order. Any near-future implementation of the LWP architecture would not have the capability to issue instructions out-of-order. However, the LWP architecture allows multiple outstanding memory accesses, which may complete in any order and thus allowing a wide range of memory consistency models requiring fully strict to fully relaxed ordering constraints.

Given a set of relaxations in the program order, the compiler can use the single thread consistency model to reason about the correctness of the programs execution within the single thread. This model also provides the compiler with the information about the possible outcomes of different sequences of memory operations. The compiler then can enforce some specific ordering constraints depending on the requirement. For example, the compiler can insert STE/STAM (see Section 4.2.2) instructions at specific points in the code to enforce uniprocessor data dependencies.

4.2.1 Single thread Consistency Model

Reasoning about the order completion of memory access falls in two categories: order of completion of instructions issuing the accesses and the order of completion of access at memory. This distinction arises due to a number of architectural features, for example, the nature of general interconnect that provides no guarantees on the order in which packets may reach the destination. The four different categories of completion of accesses is listed in the left-hand column of Table 4.2.1. We use these categories to differentiate the outcomes of the different sequences of memory accesses later in Table 4.2.1.
TABLE 4.3

DESCRIPTIONS OF SYMBOLS USED IN TABLE 4.2.1

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>II</td>
<td>Instruction complete in-order of issue</td>
</tr>
<tr>
<td>IO</td>
<td>Instruction complete out-of-order of issue</td>
</tr>
<tr>
<td>MI</td>
<td>Memory may complete operations in-order of issue</td>
</tr>
<tr>
<td>MO</td>
<td>Memory may complete operations out-of-order of issue</td>
</tr>
</tbody>
</table>

We proceed as follows: first we consider all possible combination of synchronized memory instructions within a single thread. The register and the memory address of the first memory instruction is labeled r1 and M1. Similarly, for the second instruction the register and the memory addresses is labeled r2 and M2 respectively. Now, we investigate the order of issue and completion of these instructions. Some of these instruction sequences may have implicit ordering constraints or may require external memory events for completion. We investigate these cases for cases for same and different memory addresses ($M1 = M2$ and $M1 \neq M2$) and for same or different source and target registers ($r1 = r2$ vs $r1 \neq r2$).

Some of these sequences may never be generated by the compiler, but are included here for the sake of completeness. Further, we also consider cases where there is a potential for an hazard if strict program order is not maintained. For these cases, we analyze if the introduction of additional instructions such as STE/STAM will resolve these hazards.

As mentioned earlier, the extension bit of each memory Dword in the LWP memory model, introduces a wide range of extended memory states and also additional requirements to be satisfied by the load/store instructions that target these Dwords. In short, load instructions (ld.fe, ld.ff) require the state of the target memory Dword to be full at the time of the load, whereas synchronized store instructions
(sd.ef) requires it to be empty. Normal store instructions (sd.xf) perform the store irrespective of the state of the target Dword.

Table 4.2.1 lists the initial extended memory state for the target memory locations M1 and M2 used in Table 4.2.1. By constraining the possible initial extended memory states, we are implicitly enforcing the requirement that accesses issued prior to the sequences in Table 4.2.1 have completed before either of the instructions in the sequence are issued. In other words, we have restricted this model to capture the interactions between any two instructions and not the entire thread.

<table>
<thead>
<tr>
<th>Ordering type</th>
<th>Initial state of M1</th>
<th>Initial state of M2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load-to-Load ordering</td>
<td>full</td>
<td>full</td>
</tr>
<tr>
<td>Load-to-Store ordering</td>
<td>full</td>
<td>empty</td>
</tr>
<tr>
<td>Store-to-Load ordering</td>
<td>empty</td>
<td>full</td>
</tr>
<tr>
<td>Store-to-Store ordering</td>
<td>empty</td>
<td>empty</td>
</tr>
</tbody>
</table>

Table 4.2.1 lists the possible ordering of different sequences of memory accesses. Here the symbol “→” in the sequence op1 → op2 implies the program order occurrence of the memory access (or the order of issue of these accesses). The first column of Table 4.2.1, lists all possible sequences of any two consecutive loads/stores. The next two columns represent the ordering when the address Dword is the same in both the instructions, whereas the final two columns represent the ordering when the address of the Dword is different.

The following discussion revolves primarily around the behavior of the memory access involving a single memory location. The issues related to multiple memory locations would come under the more general memory consistency model. However, in both cases, the ordering of the instructions also depends on the referenced
### TABLE 4.5

**ORDERING OF TWO INSTRUCTION PAIRS**

<table>
<thead>
<tr>
<th>Memory access sequence</th>
<th>$M_1 = M_2$</th>
<th>$M_1 \neq M_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$r_1 = r_2$</td>
<td>$r_1 \neq r_2$</td>
</tr>
<tr>
<td><strong>Load-to-Load ordering</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld.ff r1,M1 → ld.ff r2,M2</td>
<td>II,MI</td>
<td>IO,MO</td>
</tr>
<tr>
<td>ld.ff r1,M1 → ld.fe r2,M2</td>
<td>II,MI</td>
<td>IO,MO</td>
</tr>
<tr>
<td>ld.fe r1,M1 → ld.ff r2,M2</td>
<td>II,MI</td>
<td>IO,MO</td>
</tr>
<tr>
<td>ld.fe r1,M1 → ld.fe r2,M2</td>
<td>II,MI</td>
<td>IO,MO</td>
</tr>
<tr>
<td><strong>Load-to-Store ordering</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld.ff r1,M1 → sd.xf r2,M2</td>
<td>II,MI</td>
<td>IO,MO</td>
</tr>
<tr>
<td>ld.ff r1,M1 → sd.ef r2,M2</td>
<td>II,MI</td>
<td>IO,MI</td>
</tr>
<tr>
<td>ld.fe r1,M1 → sd.xf r2,M2</td>
<td>II,MI</td>
<td>IO,MO</td>
</tr>
<tr>
<td>ld.fe r1,M1 → sd.ef r2,M2</td>
<td>II,MI</td>
<td>IO,MI</td>
</tr>
<tr>
<td><strong>Store-to-Load ordering</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd.xf r1,M1 → ld.ff r2,M2</td>
<td>II,MI</td>
<td>IO,MI</td>
</tr>
<tr>
<td>sd.xf r1,M1 → ld.fe r2,M2</td>
<td>II,MI</td>
<td>IO,MI</td>
</tr>
<tr>
<td>sd.ef r1,M1 → ld.ff r2,M2</td>
<td>II,MI</td>
<td>IO,MI</td>
</tr>
<tr>
<td>sd.ef r1,M1 → ld.fe r2,M2</td>
<td>II,MI</td>
<td>IO,MI</td>
</tr>
<tr>
<td><strong>Store-to-Store ordering</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd.xf r1,M1 → sd.xf r2,M2</td>
<td>II,MI</td>
<td>IO,MO</td>
</tr>
<tr>
<td>sd.xf r1,M1 → sd.ef r2,M2</td>
<td>II,MI</td>
<td>IO,MO</td>
</tr>
<tr>
<td>sd.ef r1,M1 → sd.xf r2,M2</td>
<td>II,MI</td>
<td>IO,MO</td>
</tr>
<tr>
<td>sd.ef r1,M1 → sd.ef r2,M2</td>
<td>II,MI</td>
<td>IO,MO</td>
</tr>
</tbody>
</table>
source/target registers. From the point of view of the single thread consistency model, it is guaranteed that accesses that reference the same register are serialized. This is because the semantics LWP instructions require the thread be stalled if the registers referenced by the instruction being issued is empty.

TABLE 4.6

MEMORY SEQUENCES WITH $M_1 = M_2$ AND $r_1 = r_2$

<table>
<thead>
<tr>
<th>Memory access sequence</th>
<th>$M_1 = M_2$ and $r_1 = r_2$</th>
<th>Implicit reordering constraints?</th>
<th>External event required for completion?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld.ff r1,M1 → ld.ff r2,M2</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>ld.ff r1,M1 → ld.fe r2,M2</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>ld.fe r1,M1 → ld.ff r2,M2</td>
<td>Yes</td>
<td>ld.ff requires an external write event</td>
<td></td>
</tr>
<tr>
<td>ld.fe r1,M1 → ld.fe r2,M2</td>
<td>Yes</td>
<td>Second ld.fe requires an external write event</td>
<td></td>
</tr>
</tbody>
</table>

Load-to-Load ordering

Store-to-Store ordering

| sd.xf r1,M1 → sd.xf r2,M2 | Yes                          | No sd.ef requires an external synchronized read event |                                        |
| sd.xf r1,M1 → sd.ef r2,M2  | Yes                          | No                                |                                        |
| sd.ef r1,M1 → sd.xf r2,M2  | Yes                          | Second sd.ef requires an external synchronized read event |                                        |
| sd.ef r1,M1 → sd.ef r2,M2  | Yes                          | No                                |                                        |

Table 4.2.1 and Table 4.2.1 list some issues on ordering of instructions with $r_1 = r_2$ and $r_1 \neq r_2$ respectively. Both these tables present the information in the following three categories:

- Are there any implicit constraints on reordering operations? This refers to cases where the second instruction is not issued until the first one completes
either because the initial extended memory state is not as expected by the
second instruction or because the second instruction references the empty re-
response register of the first instruction. In table 4.2.1, all entries have an “yes”
because of the latter reason.

• Does the sequence require an external event for completion when operations
are not reordered? This deals with cases where the second instruction requires
an external event from another thread for the sequence to complete. This is
probably not code that would typically be generated by a compiler except
under very specific circumstances.

• Any violations in ordering constraints when reordered? This deals with unipro-
cessor data dependency restrictions that have to be enforced for correct pro-
gram execution. Again in table 4.2.1, none of the sequences can be reordered
because the second instruction cannot be issued until the first completes and
sets its response register to a non-empty value.

The second category listed above needs some additional explanation and is in
some ways related to the other two. We call the case when the second instruction
in program order requires an external event (synchronized/un-synchronized store
or synchronized load instruction) for completion of the sequence as a Multithread
synchronization problem. This implies that the second instruction has to wait for
another synchronizing operation to set the extended memory state to some other
state.

Things gets complicated when we reason about hazards and conditions that
violate uniprocessor data dependency restrictions. For this reason, we have included
an additional level of detail in identifying data hazards, as shown in Table 4.2.1. The
idea of categorizing hazards depending on whether they introduce the multithread
TABLE 4.7

DATA DEPENDENCY HAZARDS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAR</td>
<td>Read after read hazard</td>
</tr>
<tr>
<td>RAW</td>
<td>Read after write hazard</td>
</tr>
<tr>
<td>WAR</td>
<td>Write after read hazard</td>
</tr>
<tr>
<td>WAW</td>
<td>Write after write hazard</td>
</tr>
<tr>
<td>RAR+</td>
<td>RAR hazard introduces a multithread synchronization problem</td>
</tr>
<tr>
<td>WAW+</td>
<td>WAW hazard introduces a multithread synchronization problem</td>
</tr>
</tbody>
</table>

synchronization problem is only for the sake of completeness. If they do then the first instruction would be the one that would require the external event for the sequence to complete. As explained in the next subsection, if the hazard is avoided so is the multithread synchronization problem that arises due to the hazard.

Both Table 4.2.1 and Table 4.2.1 refer to RAR (or RAR+) hazards which is uncommon in most architectures. This primarily implies that when two load instructions are reordered, then cases may arise where the first instruction in the program order gets a value from a later store. This is primarily due to the extended memory semantics of the LWP architecture.

4.2.2 Enforcing ordering using STE/STAM instructions

The LWP ISA includes instructions that permit two kinds of ordering enforcement. The STE instruction check if some response register is non-empty, whereas, STAM checks for matching addresses. These instruction are explained below.

The *Stall on Empty (STE) instruction*, is used to test the state of a register and suspend the thread if that register is “empty”. Once the thread continues execution beyond that instruction, the thread is guaranteed that the register has a value (or at least an error response that indicates the completion of the associated memory oper-
### TABLE 4.8

MEMORY SEQUENCES WITH $M1 = M2$ AND $r1 \neq r2$

<table>
<thead>
<tr>
<th>Memory access sequence</th>
<th>$M1 = M2$ and $r1 \neq r2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Implicit reordering constraints?</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Load-to-Load ordering</strong></td>
<td></td>
</tr>
<tr>
<td>ld.ff r1,M1 → ld.ff r2,M2</td>
<td>No</td>
</tr>
<tr>
<td>ld.ff r1,M1 → ld.fe r2,M2</td>
<td>No</td>
</tr>
<tr>
<td>ld.fe r1,M1 → ld.ff r2,M2</td>
<td>No</td>
</tr>
<tr>
<td>ld.fe r1,M1 → ld.fe r2,M2</td>
<td>No</td>
</tr>
<tr>
<td><strong>Load-to-Store ordering</strong></td>
<td></td>
</tr>
<tr>
<td>ld.ff r1,M1 → sd.xf r2,M2</td>
<td>No</td>
</tr>
<tr>
<td>ld.ff r1,M1 → sd.ef r2,M2</td>
<td>Yes</td>
</tr>
<tr>
<td>ld.fe r1,M1 → sd.xf r2,M2</td>
<td>No</td>
</tr>
<tr>
<td>ld.fe r1,M1 → sd.ef r2,M2</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Store-to-Load ordering</strong></td>
<td></td>
</tr>
<tr>
<td>sd.xf r1,M1 → ld.ff r2,M2</td>
<td>Yes</td>
</tr>
<tr>
<td>sd.xf r1,M1 → ld.fe r2,M2</td>
<td>Yes</td>
</tr>
<tr>
<td>sd.ef r1,M1 → ld.ff r2,M2</td>
<td>Yes</td>
</tr>
<tr>
<td>sd.ef r1,M1 → ld.fe r2,M2</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Store-to-Store ordering</strong></td>
<td></td>
</tr>
<tr>
<td>sd.xf r1,M1 → sd.xf r2,M2</td>
<td>No</td>
</tr>
<tr>
<td>sd.xf r1,M1 → sd.ef r2,M2</td>
<td>No</td>
</tr>
<tr>
<td>sd.ef r1,M1 → sd.xf r2,M2</td>
<td>No</td>
</tr>
<tr>
<td>sd.ef r1,M1 → sd.ef r2,M2</td>
<td>No</td>
</tr>
</tbody>
</table>

74
ation). In either case, this implies that there are no outstanding memory operations in progress that uses that register as a target.

The STE instruction is a simple register-register instruction and the rationale behind the STE instruction is quite intuitive. Consider the example of a memory fence, or more specifically a full fence used to implement Release Consistency[65]. A “vanilla” implementation of the full fence would require a set of counters for keeping track of outstanding memory accesses. The “fence” is enforced by stalling the thread until the appropriate counters become zero.

This functionality can be more easily implemented in the LWP architecture, since each memory operation sets the response register to empty until it receives an acknowledgement. Thus, by testing the appropriate set of response registers the thread can determine which memory operations have completed and which have not. The thread can thus be stalled depending on the outcome of these checks.

The Stall on address match (STAM) instruction, relies on the fact that a load or a store leaves an virtual address behind in the target word. The STAM instruction computes as part of its execution a memory address in a normal fashion. The specified target register is read, and if the register is “empty” and the address stored in it matches the one just computed by the STAM, then the thread blocks on the register. If the state of the register is full (holds a value, access descriptor or a continuation), or if the address is different, then execution continues.

If STAM can no longer determine the address, then this also stalls the thread until the operation completes. This occurs when the virtual address is lost and the register is left “empty”, when some other event such as a pending thread overwrites the address with other information (for example a frame address). Unlike the STE instruction, the STAM instruction stalls the thread only when there is an actual address match and avoids some unnecessary stalls.
### Table 4.9

**Enforcing Ordering with STE/STAM Instructions**

<table>
<thead>
<tr>
<th>Memory access sequence</th>
<th>$M_1 = M_2$ and $r_1 \neq r_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Violations when reordered?</td>
</tr>
<tr>
<td><strong>Load-to-Load ordering</strong></td>
<td></td>
</tr>
<tr>
<td>ld.ff r1,M1 → ld.ff r2,M2</td>
<td>No</td>
</tr>
<tr>
<td>ld.ff r1,M1 → ld.fe r2,M2</td>
<td>RAR+</td>
</tr>
<tr>
<td>ld.fe r1,M1 → ld.ff r2,M2</td>
<td>Yes</td>
</tr>
<tr>
<td>ld.fe r1,M1 → ld.fe r2,M2</td>
<td>RAR+</td>
</tr>
<tr>
<td><strong>Load-to-Store ordering</strong></td>
<td></td>
</tr>
<tr>
<td>ld.ff r1,M1 → sd.xf r2,M2</td>
<td>WAR</td>
</tr>
<tr>
<td>ld.ff r1,M1 → sd.ef r2,M2</td>
<td>No</td>
</tr>
<tr>
<td>ld.fe r1,M1 → sd.xf r2,M2</td>
<td>WAR</td>
</tr>
<tr>
<td>ld.fe r1,M1 → sd.ef r2,M2</td>
<td>No</td>
</tr>
<tr>
<td><strong>Store-to-Store ordering</strong></td>
<td></td>
</tr>
<tr>
<td>sd.xf r1,M1 → sd.xf r2,M2</td>
<td>WAW</td>
</tr>
<tr>
<td>sd.xf r1,M1 → sd.ef r2,M2</td>
<td>WAW</td>
</tr>
<tr>
<td>sd.ef r1,M1 → sd.xf r2,M2</td>
<td>WAW+</td>
</tr>
<tr>
<td>sd.ef r1,M1 → sd.ef r2,M2</td>
<td>WAW+</td>
</tr>
</tbody>
</table>

Table 4.2.2 shows the effect of enforcing specific ordering using STE/STAM instructions which eliminate the data dependency hazards. The compiler may use this information to enforce the required ordering by inserting STE/STAM instructions. Nevertheless, it should be noted that STE/STAM instructions do not eliminate the multithread synchronization problem and more than one such instruction may be necessary when considering more than two instructions. The second column in table 4.2.2 is reproduced from table 4.2.1 for the sake of clarity of presentation.

#### 4.2.3 Compiler-view of instruction ordering

In reality, a compiler may not be able to perfectly disambiguate memory addresses i.e., determine at compile-time if any two addresses refer to the same location.
Figure 4.8. Compiler’s view of instruction ordering

in memory. In the previous sections we presented the hardware view of the execution of two-instruction sequences and relied on perfect knowledge of the memory addresses being accessed. However, this may not be the case for a compiler since it may not be able to perfectly disambiguate memory addresses.

In Table 4.2.2, we had listed nine situations where the ordering had to be enforced by explicit STE/STAM instructions between the conflicting pairs of memory operations. It should be noted that, the compiler with its limited knowledge of memory addresses has to conservatively generate the above mentioned code sequences (with the STE/STAM instructions).

The graphical representation of the compiler’s view of the instruction ordering is shown in Figure 4.8. Three situations may arise here: one, the compiler is able to perfectly disambiguate memory addresses and it determines the addresses are different, in which case the compiler does not need to generate any special code. In
the second situation, the compiler determines the addresses are the same, in which case it inserts STE (or STAM) instructions depending on whether the sequence falls in one of the nine cases discussed previously.

Finally, if the compiler is unable to disambiguate the addresses, then it can insert a STAM (or STE) instruction. Inserting a STAM may be advantageous here because the thread need not unnecessarily stall if it determines at run-time the addresses are different. It should also be noted that the above guarantees on ordering applies to enforcing consistency within a single thread and requires further treatment before it can be applied to the overall memory consistency model.

In the above discussions, we did not explicitly reason the advantages of inserting STE over STAM or vice versa. However, such a study would be necessary when we consider longer sequences of instructions because the differences in the semantics of both the instructions becomes more visible. Extensive code analysis and simulations have to be performed to evaluate the advantages of inserting one over the other.

4.3 Summary

The first part of this chapter presented a brief overview of the LWP architecture. We described the LWP, EMU, LPC and system level components briefly. Details on the thread model and the memory model was also provided. The second part of this chapter dealt with two instruction ordering model for the LWP architecture. This model investigated the necessary ordering requirements for any two memory operations within a single thread.
CHAPTER 5

SCALABLE SYNCHRONIZATION FOR LWP SYSTEMS

Scalable synchronization mechanisms have been proposed for a variety of hardware and software architectures in the past. In this chapter we investigate the constructing these scalable synchronization mechanisms based on Full/Empty bits available in the LWP hardware.

There are two kinds of LWP-based systems we investigate for evaluating these scalable mechanisms. The first kind of LWP based system would be a homogeneous system that consists of multiple LWPs distributed among multiple LPCs. We call this an All-LWP system. We use SPLASH2 benchmark and few hand coded microbenchmarks to evaluate the performance of the scalable mechanisms against a baseline SMP system. The second kind of system is a heterogeneous system where a conventional host processor “controls” a number of LWP components. Such a system can exploit the advantages of both conventional HWPs as well as highly multithreaded LWP system. We use NAS benchmarks to evaluate the performance of the synchronization mechanisms against a baseline CMP system.

The rest of this chapter is organized as follows: first we explain the two system models of interest, namely the heterogeneous LWP system and All-LWP system. We then give a brief overview of the SST simulator used to perform all the experiments presented in this chapter. We also present the SST simulation methodology which consists of runtime library setup, benchmark compilation process and the various
components and architectural parameters of the simulator. Finally we present the results and graphs of our analysis.

5.1 Using OpenMP to explore Heterogeneous architectures

The basic idea of LWP based Heterogeneous Chip Multiprocessors (CMPs) and OpenMP programming model for such systems was first proposed in [67] and further explored in [68]. In this section we evaluate the effect scalable synchronization mechanisms for such a system. First we present a brief overview on the proposed architecture of the heterogeneous CMP system. Then we describe the OpenMP programming model (Section 2.3.3) and its applicability in a heterogeneous system. Readers who are interested in the idea of exploring LWP based heterogeneous CMPs must refer to [67] and [68].

5.1.1 LWP based Heterogeneous CMP design

The traditional ways of improving performance of a single core using techniques such as increasing clock, adding more functional units, increasing cache size, increasing issue width etc are no longer effective and are facing diminishing returns. The direction which both the industry and academia are currently pursuing is based on Multicore processors or Chip Multiprocessors (CMPs). In [68], the author explores the cost/performance tradeoffs of a conventional CMP design over a novel design LWP based heterogeneous processor design. The figures 5.1 and 5.2 show the organization of the baseline CMP design and the proposed LWP based heterogeneous architecture.

The baseline system is a single CMP chip connected to DRAM over a bus. Each core in the CMP design is a conventional multiple issue pipelined superscalar processor capable of speculative execution and dynamic branch prediction. The cores have separate L1 cache but may share a L2 cache, and a cache coherency protocol
Figure 5.1. Baseline CMP System

Figure 5.2. Proposed LWP based Heterogeneous Processor System
keeps the caches coherent. For all our experiments we assume the microarchitecture of the cores are similar to an AMD Opteron Processor. The exact architectural parameters used in our simulations are presented in Section 5.5.2.

The heterogeneous system consists of a single conventional processor connected to a combination of LPC and DRAM chips. For cost purposes, only some of the DRAM chips are enhanced with LWP capability whereas the rest are plain DRAM chips. As previously explained, LPCs primarily house a number of LWP cores which are integrated with some form of DRAM memory, in our case EDRAM. The architectural parameters of the conventional processor are similar to that of individual cores in the baseline CMP design. On the other hand, the LWPs are simple single-issue pipelined processors with no branch prediction or speculative execution capabilities. Further these LWP cores support low cost multithreading in hardware and the memory supports FEB for fast producer-consumer synchronization as already explained in previous chapters.

5.1.2 OpenMP programming model

In [67] and in [68], the cost/performance tradeoffs of the proposed architecture is explored using the OpenMP programming model. The OpenMP model is well suited for the proposed heterogeneous architecture since we can assign “slave” threads to LWPs and the “master” thread to the HWP (conventional processor). The “Master” thread execute serial sections of the code and schedules parallel loop-level work to the “slave” threads.

The overall architecture and the corresponding programming model combines the best of both conventional processors and PIM enhanced memory. The serial codes executing in the conventional processor tend to have good temporal and spatial locality of the cached data and perform more branches. Contrastingly, loops tend
to be memory intensive and contain relatively fewer branches. In this thesis, we explore the benefits due to performance only. The idea of performing an integrated cost performance study make sense in the general sense but becomes too complicated for evaluating synchronization mechanisms.

5.2 All-LWP system

As explained in the beginning of this chapter, the other possible architecture we consider is an All-LWP system. Such as system consist of a number of LPC components connected to other LPC components and possibly some DRAM memory. Figures 5.3 and 5.4 show the organization of the baseline SMP design and the proposed All-LWP architecture.

This architecture has a potential to be scaled to a large number of components. Further applications that are memory intensive will have higher performance. However, single thread performance may not be as good as conventional cores given the simplicity of a single LWP code. Conventional cores tend to have better single thread performance than LWP cores due to higher clock rates, support for branch
prediction and speculative execution etc. Eventually some combination of conventional and LWP based cores may be necessary for a large scale system in order to achieve a balanced memory system along with good single thread performance.

5.3 Structural Simulator Toolkit (SST)

The Structural Simulator Toolkit (SST) [69] is a system-level simulation tool based on Enkidu discrete event simulation framework[70]. The simulator consists of three parts: the Frontend, the Backend and finally the Framework itself. We give a general overview of the simulator in the following sections. We also provide details on how SST was used in our experiments in the following sections. For more details on the internals of the toolkit refer to [69] and [70].
5.3.1 Frontends

The Frontend consists of multiple ISAs and other inputs such as traces to represent the applications that drives the simulation. In other words the simulator can simulate application binaries and traces generated for the supported frontends. For all our experiments we used a variant of the PowerPC ISA frontend with extensions to utilize special hardware capabilities such as FEB (Full/Empty Bit) mechanisms.

The SST’s PowerPC frontend supports a version of standard C library (libc). A highly modified version of the POSIX Threads (Pthreads) Library is also available as a part of SST’s libc. Most importantly, synchronization routines such as (pthread_mutex_* and pthread_cond_*) are modified to use full/empty bits. These capabilities enables users to take legacy codes written in C/C++ and compile them using a standard C compiler (gcc for MacOS X/PowerPC) to generate binaries for the PowerPC frontend. The compilation process only requires minimal variation since one should statically link the SST version of the standard C libraries to produce these binaries instead of the default system libraries.

The other interesting feature that can be exploited is the syscall (sc) mechanism. Instead of modifying a compiler to generate new instructions etc., the syscall mechanism can be used to exploit novel architectural capabilities. This is done by writing inline assembly to call the PowerPC sc instruction with appropriate arguments. When the simulator sees an sc instruction in the application’s binary, it uses the instruction’s input arguments to simulate the appropriate operation. For example, synchronized loads and stores are supported in SST by overloading the syscall mechanism.
5.3.2 Backends

The Backend of the simulator consists of program components which represent
the various architectural objects of a real system such as the LWPs, Memory chips
and the network. The use of a component-based modular approach enables the
designer to construct the architectural objects incrementally and hierarchically de-
dpending on the required level of complexity/detail. The Backend is actually the
core of the simulator since it defines the operations of the simulated architecture
and how the frontend binaries needs to be executed.

For systems based on conventional processors, SST uses a modified simplescalar
based backend. For LWP based systems, SST supports a wide range of architectural
components such as LWP core, LPC chip, Parcel handler, NIF, DRAM and EDRAM
to name a few.

5.3.3 Framework

SST uses a flexible simulation framework based on the Enkidu to tie the fro-
nend(s) and the backend(s) of the simulator and provide user services. The frame-
work is responsible for providing support of discrete event simulation, facilities
for configuring the architectural parameters, support for debugging and collecting
statistics. Further, Enkidu keeps track of “concurrent events” and hence provides
the idea of concurrency within the simulation framework.

5.4 Scalable Synchronization Mechanisms

The primary motivation to evaluate scalable synchronization mechanisms for
large scale LWP based system are twofold. First, the LWP cores we are assum-
ing are capable of supporting a large number of light weight threads in hardware.
This implies that depending on the application the frequency and overhead of syn-
chronization becomes usually high. Secondly, the FEB (Full/Empty Bits) supports lightweight producer-consumer communication, but is not efficient when multiple producers or consumers want to communicate. Additional support either in hardware and/or in software is necessary. Thus scalable mechanisms that are based on FEB are necessary to support low cost synchronization amongst the large number of threads. Further, in the context of OpenMP codes, “slave” threads have to perform frequent barrier synchronizations across loop iterations. Thus it becomes important to quantify and explore scalable synchronization mechanisms in this context.

In order to evaluate our synchronization mechanisms, we made some modifications to the Pthreads library that is part of SST’s standard C library. The two major additions are as follows. First we added a new set of calls under `pthread_barrier` to the Pthreads library to experiment with scalable barrier algorithms. Before these additions users had to use a combination of `pthread_mutex` and `pthread_cond` calls to implement a barrier. It should be noted that the `pthread_barrier` are not compiled as part of the C library but is compiled into a separate library. This is primarily to ease the process since adding new calls to libc seemed non-trivial compared to just compiling the `pthread_barrier` as a separate runtime library. Next we also modified the original SST’s implementation of the `pthread_mutex` to experiment with scalable locking mechanisms.

Overall, we present results for four barrier and seven locking mechanisms for a wide range of benchmarks. The barrier algorithms includes the dissemination barrier, tournament barrier, centralized barrier and finally the tree barrier. All these algorithms perform the lowest level atomic operations using Full/Empty bits. In later sections we also compare the FEB based implementation with regular implementations such as using PowerPC Load-Locked and Store-Conditional instructions. This helps us in quantifying the advantages of FEB even in conventional systems.
Figure 5.5. Software stack and simulator components

5.5 SST Simulation Methodology

This section gives a brief overview of the SST runtime environment setup, benchmark compilation/linking setup and the SST backend architecture setup. Figure 5.5 summarizes the benchmark compilation process, runtime library and the various simulator components.

5.5.1 Baseline CMP and SMP System

The baseline CMP and SMP designs simulated by SST are shown in figures 5.1 and 5.3 respectively. Each processor core supports a PowerPC ISA and the microarchitectural parameters are modeled to represent an AMD Opteron core. A simple snooping cache coherency protocol based on [71] ensures the caches are kept coherent.
The simulated SMP system consists of one to sixteen processors connected to each other using a bus interface. The simulated CMP system is very close to the SMP system and consists of one to sixteen cores per chip. The main difference is in the complexity of the processor core itself and the assumption that the cores in a CMP are physically on the same chip. Usually CMP core have individual L1 caches and share a large L2 cache. However, the CMP cores have separate L1 and L2 caches in our simulations.

It should be noted that both conventional cores and LWP cores use the same C Library. This implies that the lowest level synchronization routines performed by either of these processors is implemented using FEB based mechanism. The mechanism for LWP core involves blocking the thread and is listed below. However for conventional cores, the processors spin-wait on the full-empty bit. In other words, they resend memory requests to the memory system until the status of FEB changes. In the later part of this chapter, we compare a more conventional implementation of synchronization routines using PowerPC Load-Locked and Store-Conditional instructions against the FEB implementation.

5.5.2 Simulated Architecture setup and parameters

The simulated LWP architecture is similar to the one described in Chapter 4. Each LWP core consists of a simple four stage pipeline. For branch instructions, the processor stops issuing instructions from the thread until the branch is safely resolved. In other words there is no support for speculative execution or branch prediction hardware. For all our experiments, each LWP has access to 8-way 4KB instruction cache.

The simulated memory system supports FEB as part of each addressed memory word. Unlike the classical LWP implementations discussed in previous chapters, the
FEB bits can exist only in either Full or Empty state. Threads block on memory
locations when the FEB status is not the same as expected. When multiple threads
block on a single memory location they form a queue of requests. When the appro-
priate message arrives to set the FEB of the location then all the threads are woken
up. This mechanism is assumed to be supported by the hardware and completes in
one cycle.

Remote memory operations can be satisfied either by thread migration or by
sending a parcel with the instruction to the target LWP. In case of thread migration,
part of the thread’s state is packed into a parcel and sent to a remote LWP. When
a parcel with an instruction is sent, the issuing thread is blocked locally. When
the instruction completes on the remote LWP, a parcel is sent back to the LWP
on which the thread block. This parcel wakes up the blocked thread which then
resumes its execution.

The simulated threads in SST are closer to conventional threads in the sense
that they have stacks. Further each thread can have only one outstanding memory
request. When a thread gets migrated, its stack gets left behind. This implies that
future references by the migrated thread to its stack would need additional migra-
tion. Initial experiments have shown that thread migration by itself does not give
a performance benefit due to a thread’s repeated access to its stack. SST currently
supports other mechanisms such as a stack cache that boost the performance of
migrating threads. Exploring such mechanisms are currently beyond the scope of
this thesis.

The main modification to SST was to add a mechanism to enable LWP cores
to spawn threads to other LWP cores. Originally LWP cores could only fork new
threads which would run on the same core. This was necessary to provide support
for All-LWP systems. We use a round-robin based allocation strategy to distribute
### Table 5.1

**Processor Simulation Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Baseline SMP/CMP</th>
<th>All-LWP / LWP based CMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>4, 8, 16</td>
<td>8, 32, 64</td>
</tr>
<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>64K/2</td>
<td>4K/8 (I)</td>
</tr>
<tr>
<td>L2</td>
<td></td>
<td>none</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>2GHz</td>
<td>500-1000MHz</td>
</tr>
<tr>
<td># Outstanding memory requests</td>
<td>1</td>
<td>1</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>

The threads amongst all the LWPs in the system. Table 5.5.2 lists the configuration parameters for the various systems that are simulated.

#### 5.5.3 Runtime Library Compilation Process

The support for the standard C libraries in SST enabled us to compile and test a wide range of benchmarks which includes SPLASH2 and NAS NPB. In order to compile SPLASH2, we needed to provide some sort of support for the PARMACS macros. In our case, the definition for these macros is based on the Pthreads library. In other words, the PARMACS calls are converted into appropriate Pthread calls by a macro preprocessor. Our implementation of the PARMACS definitions for SST is based on [60]. We had to modify the original definitions to suit SST’s frontend and also had to change the definitions of the macros BARDEC, BARINIT and BARRIER to call the appropriate `pthread_barrier` calls. The original implementation
used a combination of `pthread_mutex` and `pthread_cond` calls to implement these macros.

SST also provides support for the OpenMP runtime environment which enables us to compile and simulate the NAS parallel benchmarks. The OpenMP API is a shared memory parallel programming model focusing on loop level parallelism. The OpenMP standard is currently supported in a wide range of architectures and operating systems. The programmer starts from a serial implementation of a code in C, C++ or Fortran and adds `#pragma` annotations indicate parallel sections and synchronization operations to the source code.

For all the experiments presented in this section, we used the Omni OpenMP (version 1.6) [72] compiler that is a source-to-source translator. The Fortran and C sources from the NAS codes were first preprocessed using the Omni OpenMP frontend. The Omni OpenMP compiler produces C code with calls to the Omni Runtime library. For these experiments, an Omni OpenMP runtime implementation using Pthreads was used. In other words the Omni OpenMP runtime calls were wrappers around the appropriate Pthread calls which were already implemented for SST as mentioned above. Finally the Omni OpenMP runtime library was compiled for SST similar to the standard C library or the `pthread_barrier` library. Thus for NAS parallel benchmarks or any other OpenMP codes one had to link the Omni OpenMP runtime library to compile/test the codes.

There are two additional details that have to be noted for the Omni OpenMP runtime implementation. First, the Omni OpenMP runtime uses an internal barrier data structure to support both the implicit (compiler-generated) or the explicit (user-specified) OpenMP barrier synchronization. We modified this to use the appropriate `pthread_barrier` calls primarily to leverage the scalable implementations mentioned above. Second, the Omni OpenMP runtime was modified (by Arun
TABLE 5.2

SPLASH2: SIMULATED PROBLEM SIZE

<table>
<thead>
<tr>
<th>Kernel/Application</th>
<th>Simulated problem size</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>$2^{18}$ points</td>
</tr>
<tr>
<td>LU (Contiguous)</td>
<td>512x512 matrices</td>
</tr>
<tr>
<td>LU (Non-contiguous)</td>
<td>512x512 matrices</td>
</tr>
<tr>
<td>Radix</td>
<td>262,144 integers, radix 1024</td>
</tr>
<tr>
<td>Ocean-contiguous</td>
<td>258x258 grid</td>
</tr>
<tr>
<td>Ocean-noncontiguous</td>
<td>258x258 grid</td>
</tr>
</tbody>
</table>

Rodrigues) to account for the heterogeneous nature of the target system under consideration. Details on the exact modifications to the work partitioning algorithm can be found at [68].

5.5.4 Benchmark Compilation Process

All benchmarks listed below were compiled using a PowerPC MacOS X gcc compiler (version 3.3) and statically linked to the appropriate SST version of the libc, pthread_barrier and/or Omni OpenMP runtime library (only for NAS parallel benchmarks).

The microbenchmarks (presented in Section 3.1) were written using the Pthreads library and were compiled and statically linked with libc, pthread_barrier library for SST. The SPLASH2 kernel/application codes were first preprocessed by the m4 macro processor to convert the PARMACS calls to corresponding Pthread calls. Next, the Pthread based sources were compiled/linked as in the case of the microbenchmarks. The input data sizes for SPLASH2 codes is listed in table 5.5.4.

Finally, the NAS Fortran and C codes were first processed by a MacOS X version of the Omni OpenMP frontends to convert the OpenMP hints to Omni OpenMP
runtime calls. These codes were then statically linked using *libc, pthread_barrier* and the Omni OpenMP runtime library. Due to compiler issues we were unable to get EP, IS and UA of the NAS parallel benchmark suite to run on SST. For all other NAS codes we used the data set size S in all our experiments.

For the sake of clarity we list below the exact list of contributions to the application side of the simulation setup.

- Added four barrier and seven locking mechanisms to Pthreads library under *pthread_barrier* and *pthread_mutex* calls respectively.

- Modified the original implementation of the PARMACS definitions to suite SST and also made it call the appropriate *pthread_barrier* calls for macros BARDEC, BARINIT and BARRIER.

- Coded, compiled and tested all the microbenchmarks listed above.

- Compiled and tested all the SPLASH2 applications listed above.

- Compiled and tested the Omni frontend tools for MacOS X.

- Modified the Omni runtime library to make it call the appropriate *pthread_barrier* calls. It should be noted here that the Omni runtime library was compiled and modified (for work partitioning algorithm) by Arun Rodrigues as mentioned above.

- Compiled and tested NAS parallel benchmarks for Omni OpenMP compiler on MacOS X.

5.6 Results

This section presents the performance results for the microbenchmarks and the applications. We measure the total number of cycles to complete the benchmark.
For the barrier microbenchmark, we also measure the average time, wake time and wait time as mentioned in section 2.6.

5.6.1 All-LWP: Microbenchmarks

Figures 5.6(a)-5.6(d) show the average time per barrier and the total execution time in cycles for the entire barrier microbenchmark. Threads perform 2000 iterations of each barrier and execute some amount of random work in between barriers. The SMP configuration consists of 4 to 16 processors each running one thread. The LWP configuration consists of one LPC with eight LWPs running 4 to 512 threads.

From figures 5.6(a) and 5.6(b), it can be observed the SMP configuration outperforms the LWP for the same number of threads. However, this is expected since the SMP processors are more powerful than its LWP counterpart. Further, with more number of threads, the average time per barrier increases logarithmically for the dissemination, tournament and tree barriers. The central barrier takes the most time as it blocks all the threads on a single shared variable. The same is true for the LWP architecture as shown in figures 5.6(c) and 5.6(d).

In general, it can be observed that increasing the number of threads increases the total execution time. This is because of two reasons. First, the average time to achieve a single barrier increases. Second, the number of number of barrier iterations remains constant. This is an artifact of scaling the problem size as the number of threads increases.

Figures 5.7(a)-5.7(d) show the average wait and wake times for the four barriers in an SMP as well as All-LWP system. These metrics follow trends similar to the average time as shown above. For small number of threads in the LWP configuration in figure 5.7(b), it can be seen that the central barrier requires lesser cycles to wake the threads than the dissemination or tree barriers. This is partly due to the
Figure 5.6. Barrier Microbenchmark: All-LWP and SMP
overheads involved in the tree based barrier algorithms. However, this also points out a limitation in the simulator i.e. threads are woken up instantaneously by the hardware. However as we increase the number of threads, there is more contention and only one thread succeeds.

Figures 5.8(a)-5.8(d) and 5.9(a)-5.9(d) shows the total execution time for four lock microbenchmarks. Threads perform 500 iterations of a loop in which they acquire a lock, update a shared data structure and then release the lock. For fairness sake, a small amount of random work is also performed within the loop after the lock is released. As in the case of the barrier microbenchmark, the problem size is scaled as the number of threads increases.

Figures 5.8(a)-5.8(d) show the total execution time for the four microbenchmarks for the SMP case. We tested five locks namely, Test-and-Set (LLSC\_TS), Test-and-TS (LLSC\_TTS), Test-and-Set with exponential backoff (LLSC\_EXPTS), Test-and-TS with exponential backoff (LLSC\_EXPTTS), and ticket lock (TICKET). All these locks were implemented using the LL and SC instructions as explained in chapter 2.

In general, the ticket lock outperforms all the other locks for 4 to 16 processors. We also performed some limited tests based on queue based locks using LL and SC instructions. Since the number of simulated processors is small, the overhead of these locks did not provide sufficient improvement in the SMP case. We do not present those numbers here.

Figures 5.9(a)-5.9(d) show the total execution time for the four microbenchmarks for the LWP case. We tested three locks namely, the default full/empty bit based lock (FEB), a queue lock based on MCS lock (QUEUE) and the ticket lock (LLSC\_TICKET). The first two locks use full/empty bits where as the ticket lock is implemented using LL and SC instructions as in the SMP case. The queue lock
(a) SMP vs. All-LWP: Wait time per barrier  (b) SMP vs. All-LWP: Wake time per barrier

(c) All-LWP: Wait time per barrier  (d) All-LWP: Wake time per barrier

Figure 5.7. Barrier Microbenchmark: All-LWP and SMP
Figure 5.8. Lock Microbenchmarks: SMP
consists of a linked list of elements with each element representing a thread. We use full/empty bits to ensure atomicity. Also, unlike the FEB lock, only one thread blocks on each memory location.

In general it can be seen that the full/empty bit based locks outperform the LLSC based ticket lock. Further, the queue based lock outperforms the FEB lock for large number of threads (64 and above). In the case of the ticket lock, it should be noted that we do not present the numbers for 512 threads for any of the microbenchmarks and 256 threads for the nuca microbenchmark. The performance advantage of queue locks is significant even when the hardware incurs no delay for waking up blocked threads.

5.6.2 All-LWP: SPLASH2

Figures 5.10(a)-5.10(d) and 5.11(a)-5.11(b) show the total execution time for the SPLASH2 benchmarks. We present the results for four kernels (FFT, RADIX, LU Contiguous and LU Non-Contiguous) and two applications (OCEAN Contiguous and OCEAN Non-Contiguous). All tests used the basic FEB lock for performing mutex synchronization.

Unlike the microbenchmarks, the problem size (table 5.5.4) for SPLASH2 codes is fixed as the number of threads increases. The LWP architecture performs best for 64 or 128 threads. This corresponds to 8 to 16 threads per LWP core given eight cores in a single LPC.

We also conducted experiments for the SMP configurations with 4, 8 and 16 processors. In general, the SMP configuration outperforms the LWP configuration for the six benchmarks tested for the same number of threads. This can be attributed to two reasons. First, each SMP processor is more powerful than its LWP counterpart.
Figure 5.9. Lock Microbenchmarks: All-LWP (1 LPC with 8 LWPs)
Second, SPLASH2 is written for shared memory systems with heavy weight processors. The code performs relatively more arithmetic and branch instructions than memory or synchronization instructions. Further, the work distribution in statically decided at the beginning of the application and threads continuously for the entire run of the benchmark.

In general, it can be observed the actual barrier algorithm does not affect the performance of the benchmark much. We believe this is again the artifact of the application being more compute intensive than memory intensive. For example, in the case of 512 threads in the FFT benchmark, only 30% of the instructions committed by each LWP were loads/stores. A big fraction of the memory instructions also includes memory instructions executed by the LWP for requests generated by threads running on other LWPs. Since LWPs have one-to-one mapping with the memory they control, requests from other LWPs need to be executed.

5.6.3 Heterogeneous System: Microbenchmarks

Figures 5.12(a)-5.12(d) shows the total execution time for three lock and one barrier microbenchmark for the CMP configurations. Unlike the SMP or All-LWP case, we keep the size of the problem constant as we increase the number of threads/processors. For the lock benchmarks, threads perform a total of 51200 updates to the shared data structure. For the barrier benchmark, threads perform a total of 12800 iterations of barrier synchronizations. In other words, we perform 3200 barrier synchronizations for the 4 threads case whereas 25 for the 512 threads case.

For the CMP case, we performed tests for the four barrier and two lock algorithms. The locking algorithms in this case are based on FEB implementations and not on LL/SC instructions. The single and multiple counter benchmarks behave as expected. Increasing the number of processors, decreases the number of itera-
Figure 5.10. SPLASH2: All-LWP (1 LPC with 8 LWPs)
tions per processor and hence the total time required to complete the benchmark decreases. In the case of the doubly list, as the number of processors increases, the contention of the lock increases and is not offset by the lesser number of iterations.

Figures 5.13(a)-5.13(d) shows the total execution time for three lock and one barrier microbenchmark for the heterogeneous LWP configuration. The figures compare the execution time for the microbenchmarks for different number of threads and for two LPC/LWP configurations. We consider a 4 LPC and a 8 LPC configuration with 8 LWPs per LPC for both cases. It should be noted that we do not present the 8 LPC results for the doubly list microbenchmark.

Similar to the CMP case, we perform experiments with a constant problem size. Depending on the benchmark, the performance peaks at 256 or 512 threads. This is as expected since we are performing less number of iterations per thread as we increase the number of threads.
Figure 5.12. Lock Microbenchmarks: SMP
Figure 5.13. Microbenchmarks: Heterogeneous LWP system (4/8 LPC with 8 LWPs each)
Figure 5.14. MG: Total Exec. time

Further, the 4 LPC configuration consistently outperforms the 8 LPC configuration. This may be due to the memory allocation policy of SST.

5.6.4 Heterogeneous System: NAS

Currently, we provide only limited results for the NAS benchmarks. Figure 5.14 shows the performance of BT benchmark (size S) for the default and tree barriers. We used four LPCs with each having eight LWP. For the heterogeneous system this translates to one HWP along with four DRAM chips and four LPC chips. We are currently in the process of analyzing other benchmarks from the NAS suite.

In general, the tree barrier outperforms the default barrier in most case but only by a very small margin. In terms of the number of threads, the benchmark performs best for 32 or 64 threads which translates to one to two threads per LWP. It should be noted that the 512 threads case outperforms the 4 threads case.
CHAPTER 6

ACTIVE GRAPH SYNCHRONIZATION MECHANISMS

An important objective of this thesis is to experiment with novel techniques for performing synchronization on the LWP architecture. The previous chapter presented some insight into scalable synchronization mechanisms for LWP based systems. The proposed techniques primarily build on top of full/empty bits and light-weight multithreading. In this chapter we extend this idea one step further to explore the ideas of Active Graphs and synchronization mechanisms based on active graphs.

We take a more radical approach in designing these algorithms. In addition to using FEB mechanisms, we also exploit extended memory semantics and thread to thread communication using register frames. Further, we explore a new language (DimC language) for writing multithreaded programs for the LWP architecture.

The rest of the chapter is organized as follows: First we present some basic concepts of active graphs. Then we describe the simulation environment that consists of DimC (DiminishedC) compiler and SALT (Simulator for the Analysis of LWP Timing) toolkit. Then we describe the simulation setup for our experiments i.e. explain about the application compilation process, simulation parameters etc. Finally we present the analysis and results for the performance evaluation of Active graph synchronization mechanisms.
6.1 Active Graphs and Data Structures

A new class of high-end applications exhibiting graph-like data structures/algorithms are beginning to overwhelm even the most modern parallel supercomputers [73]. These graphs usually tend to be very large with a dynamic collection of vertices and nodes. These graphs involve highly concurrent node centric computations and the communication is largely driven by fine grained data transfer between nodes.

The idea of Active graphs [74] has been formulated to tackle graph-based problems using concepts of light-weight multithreading and the LWP architecture. Each node in the graph is associated with a distinct thread. The communication between the nodes is intuitively represented by the producer-consumer relationship between the threads tied to the respective nodes. These structures are also capable of exploiting ideas such as direct thread-to-thread communication using memory mapped thread frames, extended memory semantics, light-weight thread spawning and management.

The execution state of each thread can be encapsulated within the confines of its thread frames and may “overflow” into memory as and when necessary. A thread’s payload may include pointers to specific locations within parent/child thread frames for communicating data values and for signaling synchronization events. In addition to this threads may also be linked together through specific registers in their frames for facilitating some thread management operations.

Figure 6.1 shows a subset of nodes in a active graph represented as thread frames. In this example, directed edges determine the direction of flow of data. Each thread’s state is arbitrarily divided into data pointers, sync pointers, working registers and thread management data. The actual pointers may be setup during the initialization phase of the computation. The straight, dashed and dotted edges represents the flow of data, synchronization and thread management signals respectively.
Figure 6.1. Active graphs
In this thesis, we will investigate a more restricted form of active graphs. The active barrier implementations we consider are of limited form in two main characteristics. First, the graphs we are dealing with are usually trees with fixed fanin and fanout for each node. Second, the actual communication “links” are static. Thus, once initialized these edges do not change during the lifetime of the data structure.

Active graphs have more potential than just represent graph-based problems. The idea in general can be viewed as a possible way to exploit software concurrency using light weight threads. As multicore processors get ubiquitous the necessity for multithreaded programming would become undeniable. A large percentage of the software community has already embarked on a new trail of exploiting concurrency in software[75][76]. This is important if the available hardware concurrency has to be exploited.

Currently, the focus is on creating threads explicitly and assigning data structures to them. The programmer, in this case, is burdened with low-level thread management as well as data structure integrity or consistency. Further, this model is not really scalable because assigning a set of objects or part of the global data structure to a thread in itself requires some sophistication. Techniques such as Active graphs allows objects to own threads and transfers the burden to the hardware or the runtime to manage them. In general, these ideas fall under a larger class of techniques that combine concepts of Object-Oriented programming languages with multithreaded programming.

6.2 Simulation Environment

This section describes the SALT/DimC simulation toolkit. We also present some details on the simulated ISA and microarchitecture. Interested readers can refer to [66] and [77] for more details on these simulation tools.
6.2.1 DimC Language and Compiler

The DimC (Diminished C) compiler [66] is used to compile programs written in the DimC language. Gary Block of Caltech/JPL is the primary developer of the DimC compiler. The DimC language is based on the C language with additional language constructs to enable the user to exploit some key ISA/architectural features of the proposed LWP architecture. For that purpose the language provides support for threaded function calls, full/empty bit synchronization and ability to access a given threads frame as part of memory. Table 6.2.1 lists some of the language extensions available for the programmer.

<table>
<thead>
<tr>
<th>Language construct</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread keyword</td>
<td>Denotes threaded function calls</td>
</tr>
<tr>
<td>Handler Keyword</td>
<td>Denotes special thread that is spawned to process exceptions in software</td>
</tr>
<tr>
<td>Release Statement</td>
<td>Allows the spawned child thread to signal parent thread to continue execution</td>
</tr>
<tr>
<td>Purge Operator</td>
<td>Set the FEB to empty for a given memory location</td>
</tr>
<tr>
<td>Prefix % Operator</td>
<td>Perform synchronized loads and stores for a given memory location</td>
</tr>
<tr>
<td>Check Operator</td>
<td>Wait on a register to get a value from an earlier memory operation</td>
</tr>
</tbody>
</table>

TABLE 6.1

PARTIAL LIST OF DimC LANGUAGE EXTENSIONS

Currently the DimC compiler supports two target ISAs which are the Tinman LWP ISA and the MTA ISA [2] and [78]. Both these ISAs are augmented with LWP specific instructions such as spawn, trip etc. The compiler backend can generate assembly output for either of the above target ISAs which then can be simulated by the SALT simulator described below. For all the experiments in this chapter we use the target MTA ISA.
The DimC compiler lacks support for a runtime library. Since there is no linker, the assembly code generated by DimC uses absolute addresses and unlike GCC, DimC does not generate a relocatable module. This also implies that the entire program must be compiled each time and all source modules must be included in the main module. Considerable care must be taken to ensure that registers are used wisely, given the limited number of registers available within a thread frame and the lack of any register spilling mechanism in DimC.

As it deviates from the standard C language in these aspects, compiling legacy codes becomes a non-trivial task since we have to rewrite the whole code. However some of its language features listed above makes it an important tool to investigate the architectures under consideration. Most importantly we were able to investigate barrier synchronization mechanisms based on active graphs and test them for simple microbenchmarks as explained below.

6.2.2 SALT Simulator

SALT [77] is an instruction-level simulator for the LWP architecture. Paul Springer of Caltech/JPL is the primary developer of the SALT simulator. The primary purpose of SALT is to experiment with multiple ISAs and architecture configurations to explore the design space for LWP designs. The internal organization of SALT is similar to SST in that it consists of multiple frontends and backends to simulate different ISAs and microarchitectures. Further SALT also uses the Enkidu [70] framework to coordinate the communication between the various backend components and models time within the simulator.

SALT has been developed in conjunction with the DimC language/compiler to model the LWP architecture. SALT models threads whose state consist of 32 64-bit register frames. Memory within an LPC is shared among all LWPs. Registers within
a frame are mapped to memory via the frame pointer. Threads are allowed to have concurrent outstanding loads and stores. However the threads try to use registers that have pending memory operations then they automatically block on them until the operation completes.

SALT parses through the assembly output of program files generated by DimC. Thus it supports a MTA frontend and a Tinman IWP frontend to simulate the corresponding assemblies. All the experiments in this chapter use the MTA frontend.

Currently SALT generates limited statistics which consists of the number of active threads and the total number of threads for each simulated cycle. SALT does not model the performance of the memory system, network etc., implying that these components have zero access latency. This zero-delay model represents the classical Parallel Random Access Memory (PRAM) model [79].

6.2.3 Software Synchronization Trap Handler

SALT/DimC takes a software approach to handle multiple producer/consumer synchronization. Currently the toolkit provides users with a synchronization trap handler. When a thread finds a conflicting state for a memory location, a software handler is spawned off to handle the request behalf of the thread. The handler thread queues the request on a software queue which is later handled when an appropriate reply arrives at the memory location.

The handler itself is written in DimC and is invoked by the simulated hardware when it identifies the need for one. The actual implementation of the handler is beyond the scope of this thesis. Interested readers must refer to the SALT/DimC documentation.
6.3 Barrier Implementations using Active Data structures

We implemented four barrier algorithms (namely central, dissemination, tournament and tree barrier) in DimC language. Each node in the dissemination, tour and tree barrier algorithms is managed by a “barrier thread”. The actual algorithms are the same as described in earlier chapters, but our implementation uses the idea of active graphs to construct the data structure and expose the available concurrency. Figure 6.2 depicts the overall active graph barrier mechanism. The figure does not show the actual barrier data structure itself.

We use a simple microbenchmark to evaluate our barriers using the SALT/DimC toolkit. The microbenchmark creates $N(= 2, 4, 8, 16, 32)$ “producer” threads. Each of the producer threads, spawns off a “barrier” thread. This operation occurs in parallel to the main thread spawning off new producer threads. Once all the producer threads have been created, the main thread waits for all the producer threads to complete by performing a thread join.

The producer threads wait for a random amount of delay and signal their respective barrier threads to begin performing the barrier synchronization. This is initiated by setting the Clear to Barrier (CTB) register in the barrier thread’s frame. This register is used by the barrier thread to block waiting for its producer thread. The producer thread then blocks itself on the Clear to Proceed (CTP) register in its own frame.

On completion, the barrier thread communicates back to its producer thread to resume its operation. This is done by setting the CTP register in the producer thread’s frame which wakes up the blocked producer thread. The barrier thread then blocks on the CTB register on its own frame until the producer thread eventually wakes it up. Each producer thread is in-sync with its corresponding barrier thread and with other producer threads.
Figure 6.2. Active Graph Barrier
The communication between the producer thread and its corresponding barrier thread as well as that between various barrier threads occur by performing a combination of synchronized loads/stores. The target of these operations may be within a thread’s frame or in memory. The actual initialization of the various pointers (CTB* and CTP*) as well as the initialization of the barrier data structure is explicitly performed in the beginning and is not shown in figure 6.2.

SALT currently has a limit on the maximum number of threads that can exist per LWP. This value is limited by the amount of storage allocated to thread frames in SALT and is currently fixed at 128. This actually creates a critical limitation when running application codes with large number of threads. Due to timing issues, a memory location may have multiple handler threads associated with it. This may happen when the software queue is currently being operated on by another handler thread and concurrent accesses to the queue may result in inconsistencies. Thus one must ensure that the sum of application threads and the handler threads eventually created does not exceed this 128.

Unlike the other barriers, the centralized barrier invokes the software EMS trap handler to satisfy multiple requests to the same memory location. The other barriers do not require the EMS handler for the most part since they perform pairwise synchronizations. For 32 barrier threads, the microbenchmark created more than 128 threads (producer, barrier and EMS handler) during the execution of the centralized microbenchmark which resulted in termination of the program.

We had to choose between redesigning the benchmarks vs. performing the tests for smaller number of threads. In the end, we decided to redesign only the centralized barrier microbenchmark. The centralized barrier microbenchmark does not create additional barrier threads to perform barrier synchronization. Instead, the producer threads perform the barrier algorithm themselves.
Two things have to be noted in this regard. First for the centralized barrier, the idea of active graphs does not really apply since there is only one memory location. Thus the modification to the microbenchmark does not affect our original intentions of constructing active graph based barriers. Second, though this may seem to benefit the centralized barriers, the results in section 6.5 contradict this. In short, the overheads of the EMS handler is much larger than spawning off additional barrier threads and making them perform the barrier algorithm.

6.4 Experimentation Setup

Figure 6.3 shows the microbenchmark compilation process using the DimC/SALT toolkit. Table 6.2 lists some of the simulation parameters used for all the experiments in this chapter. Though these numbers may not really represent real world applications, the intention of these experiments is to evaluate the concept. More rigorous studies are necessary to ascertain the complete benefits of the proposed ideas.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>frontend</td>
<td>MTA</td>
</tr>
<tr>
<td># LPC</td>
<td>1</td>
</tr>
<tr>
<td># LWP</td>
<td>1</td>
</tr>
<tr>
<td>Memory size</td>
<td>2Gb</td>
</tr>
<tr>
<td>Size of frame</td>
<td>32 64-bit registers</td>
</tr>
<tr>
<td># outstanding memory requests</td>
<td>Limited only by data dependency</td>
</tr>
<tr>
<td># threads</td>
<td>1, 2, 4, 8, 16, 32</td>
</tr>
<tr>
<td>Max. # threads per LWP</td>
<td>12</td>
</tr>
<tr>
<td># Barrier iterations</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>500</td>
</tr>
</tbody>
</table>

TABLE 6.2

SALT LWP SIMULATION PARAMETERS
6.5 Results

In this section we first describe the statistics used for evaluation of the barrier mechanisms. Then we present the analysis of the results for the different mechanisms.

6.5.1 Statistics

As already mentioned SALT provides minimal information on the program execution. However, when we use the debug mode, SALT provides an instruction trace of the program. This information mainly includes the LPC id, the opcode and the simulation cycle. We use these to determine statistics such as total cycle time, average number of active threads, the maximum number of active threads, IPC and the instruction mix of the microbenchmark. All the values are measured for the entire run of the microbenchmark performing 500 iterations of barrier synchronization.
The average number and the maximum number of active threads gives us an idea on the bounds of the frame cache. For dissemination, tournament and tree barriers, the number of active threads is determined by the sum of the barrier, producer and synchronization trap handler threads that are active. For the centralized barrier, this number is the sum of producer and EMS handler threads. Further, in all the cases, the actual value is rounded-off to the closest integer.

The instruction mix gives us the number of synchronized loads/stores (SYNC), LWP specific instructions (LWP), all memory loads/stores (MEM) in the total number of instructions (TOTAL) committed. LWP specific instructions includes trip, spawn and fork. The normal loads/stores is the difference of MEM and SYNC.

The instruction per cycle (IPC) count represents the utilization of the LWP processor. For all the experiments, we consistently observed a IPC close to one (.999) for all the barriers. This is primarily the side-effect of the zero-delay performance model for the memory, frame cache, and network etc. Thus, threads do not incur any delay when accessing memory or blocking for synchronization operations.

It should also be noted that the cycle time metric is also affected by the zero-delay model. Therefore the cycle time represents the sum of number of instructions executed (ideally one every cycle) and the null instructions. Though not shown below, the null instructions occur during the initialization part of the microbenchmark. This implies the values represented in the graph (figure 6.4) do not differ much from the total instructions committed (TOTAL field of table 6.3).

6.5.2 Analysis

Table 6.3 gives the instruction mix and the length of the active queue for the four barrier algorithms. The table is divided into five sections based on the number of barrier threads participating in the synchronization operation.
### TABLE 6.3

**ACTIVE BARRIERS: INST. MIX, AVG. AND MAX. THREADS**

<table>
<thead>
<tr>
<th>Barrier</th>
<th>SYNC</th>
<th>MEM</th>
<th>LWP</th>
<th>TOTAL</th>
<th>Avg. Threads</th>
<th>Max. Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Threads 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Central</td>
<td>2004</td>
<td>9524</td>
<td>4</td>
<td>470634</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Dissemination</td>
<td>5016</td>
<td>7088</td>
<td>12</td>
<td>436548</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Tournament</td>
<td>4026</td>
<td>11123</td>
<td>12</td>
<td>510049</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Tree</td>
<td>4017</td>
<td>23106</td>
<td>12</td>
<td>532187</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Threads 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Central</td>
<td>21184</td>
<td>32102</td>
<td>2008</td>
<td>1123831</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Dissemination</td>
<td>14036</td>
<td>20182</td>
<td>24</td>
<td>895216</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Tournament</td>
<td>9052</td>
<td>27243</td>
<td>24</td>
<td>1045133</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Tree</td>
<td>9035</td>
<td>47206</td>
<td>24</td>
<td>1046864</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>Threads 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>Central</td>
<td>20836</td>
<td>317475</td>
<td>5876</td>
<td>2504965</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>Dissemination</td>
<td>36080</td>
<td>52382</td>
<td>48</td>
<td>1834696</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>Tournament</td>
<td>19104</td>
<td>59483</td>
<td>48</td>
<td>2115302</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>Tree</td>
<td>19071</td>
<td>95406</td>
<td>48</td>
<td>2076219</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>35</td>
</tr>
<tr>
<td>Threads 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>33</td>
</tr>
<tr>
<td>Central</td>
<td>57774</td>
<td>1204222</td>
<td>19097</td>
<td>8198443</td>
<td>11</td>
<td>35</td>
</tr>
<tr>
<td>Dissemination</td>
<td>88176</td>
<td>128806</td>
<td>96</td>
<td>3757944</td>
<td>16</td>
<td>33</td>
</tr>
<tr>
<td>Tournament</td>
<td>39208</td>
<td>123963</td>
<td>96</td>
<td>4255638</td>
<td>15</td>
<td>33</td>
</tr>
<tr>
<td>Tree</td>
<td>39143</td>
<td>191806</td>
<td>96</td>
<td>4134924</td>
<td>16</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>66</td>
</tr>
<tr>
<td>Threads 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>66</td>
</tr>
<tr>
<td>Central</td>
<td>110415</td>
<td>2351283</td>
<td>37696</td>
<td>15846279</td>
<td>11</td>
<td>66</td>
</tr>
<tr>
<td>Dissemination</td>
<td>208384</td>
<td>305702</td>
<td>192</td>
<td>7693210</td>
<td>33</td>
<td>65</td>
</tr>
<tr>
<td>Tournament</td>
<td>79416</td>
<td>252923</td>
<td>192</td>
<td>8536504</td>
<td>31</td>
<td>65</td>
</tr>
<tr>
<td>Tree</td>
<td>79287</td>
<td>384606</td>
<td>192</td>
<td>8252352</td>
<td>32</td>
<td>65</td>
</tr>
</tbody>
</table>
Figure 6.4. SALT/DimC: Barrier microbenchmark

Figure 6.4 presents the cycle time for the different barrier algorithms. For two threads, the centralized barrier outperforms the tree and the tournament barriers in all metrics. However, this does not appear to be the trend when more number of threads are participating in the barrier. For example in the case of 32 threads, the centralized barrier executes almost twice the number of instructions as the other barriers. As we increase the number of threads, the contention for a single memory location and the overheads of the EMS handler thread increases. This clearly shows the need for scalable synchronization mechanisms for the LWP architecture.

In general, the dissemination barrier outperforms all the other barriers based on cycle time and total instructions committed. It requires $7-10\%$ less cycles than the tree and the tournament barrier. However, it should be noted that the dissemination barrier executes more SYNC instructions than all other barriers. These instructions are the costliest compared to memory or arithmetic instructions since they block the
issuing threads for long periods of time. A more sophisticated performance model may change this behavior.

The average number of active threads for the dissemination, tournament and tree barriers is close to the total number of producer or barrier threads available. However, the maximum number of threads in the active queue is equal for a given set of barrier threads. This is an artifact of the microbenchmark. For example, the maximum number of active threads for 16 barrier threads is 33. This is because each barrier thread has a corresponding parent producer thread and all the producer threads are spawned off by a single main thread.

The centralized barrier exhibits a different trend and is not as straightforward. The average or the maximum number represents the sum of producer and EMS threads that are being spawned off for handling multiple producers/consumers. Specifically, for the 32 thread case, the maximum number of active threads is 66. This indicates a situation where each producer thread has spawned off at least one other EMS handler thread and all these threads are trying to update the software queue. The higher number of SYNC instructions executed for the centralized barrier further emphasizes the role of handler threads. In all other cases, each thread performs only 6 LWP instructions. It should be noted that the software handler can avoided by using AMOs directly instead of invoking the software handler.
CHAPTER 7

F.A.S.T: FUTEX AWARE SYNCHRONIZATION TECHNIQUES

A number of prior research efforts have investigated various thread scheduling mechanisms in order to allow for the reuse of the processor cache. In this chapter, we investigate a scheduling mechanism that assigns threads based on the shared locks. The locality of the critical section data can be exploited by enforcing an affinity between locks and the processor that has cached the execution state of the critical section protected by that lock.

We investigate the idea of migrating threads to the processor that already hold the lock and has cached the critical section data in its local cache, referred here to as the lock hot processor. This enables threads to reuse the critical section data from the processors cache and release the lock faster for other threads. We argue that this mechanism should improve the scalability of performance for highly multithreaded scientific applications.

We test our hypothesis on a 4-way Itanium2 SMP running the 2.6.9 Linux kernel. We modified the Linux 2.6 kernels O(1) scheduler using information from the Futex (Fast User-space muTEX) mechanism in order to implement our policy.

Using synthetic micro-benchmarks, we show 10-90% performance improvement in CPU cycles, L2/L3 miss ratios and number of bus requests for applications that operate on significant amounts of data inside the critical section protected by locks.
Using SPLASH2 application suite we also show that some benchmarks exhibit little or no performance improvement.

The remainder of the chapter is organized as follows: first, we present details on scheduling and synchronization techniques of the Linux Kernel. Second, we explain our modifications to the Linux kernel scheduler to implement our policy. Next we describe the experimental setup and present the results of our evaluation. The work presented in this chapter was done in collaboration with Brett Keck of the Department of Computer Science and Engineering at Notre Dame.

7.1 Introduction

Scheduling threads on shared memory multiprocessors (especially SMPs) based on cache affinity has been previously investigated in [80], [81], [82] and [83]. These proposals try to efficiently reuse the thread’s state that is already in a processor’s cache by enforcing an affinity between the processor and the threads executing on them.

In this chapter, we apply this idea to locks and data in critical sections protected by these locks. The locality of the critical section data can be exploited by enforcing an affinity between locks and the processor that has cached the execution state of the critical section protected by that lock. We investigate the idea of migrating threads to the “lock hot” processor, enabling the threads to reuse the critical section data from the processor’s cache and release the lock faster for other threads. We argue this mechanism should improve the scalability of performance for highly multithreaded scientific applications.

The main advantage of this approach is that since we are enabling threads to be closer to the execution state accessed within a critical section, this technique minimizes costly cache misses and improves the cache locality awareness of multi-
threaded scientific applications. Further, moving threads that do not require the lock away from the processor enables the thread holding the lock to complete its execution faster and release the lock quicker for other contending threads. Finally, reduction in cache misses also translates to reduction in number of requests on the communication network (shared bus in the case of small scale SMP) and hence can improve scalability by adding more processors to the system.

In general, this idea can also be viewed from a LWT standpoint. Given the small state of light-weight threads and the “closeness” to memory, light-weight threads can be migrated to the location of data. This is in contrast to “von-Neumann” based computing where the data is fetched from memory to site of processing. The concept of “traveling threads” [84] tries to achieve this. The same mechanism can also be achieved by creating a thread in the remote location, killing the parent thread and continuing execution in the remote location. In our case, threads are migrated on synchronization points to other processors in an SMP system, instead of moving the lock’s cache line and critical section data to the first processor.

To test our hypothesis we used a 4-way Itanium2 SMP running the 2.6.9 Linux kernel. We have extensively leveraged the Linux 2.6 kernel’s O(1) scheduler and Futex (Fast User-space muTEX) mechanism in order to implement our ideas. The O(1) scheduler, which takes constant time irrespective of the number of threads and the Futex mechanism, which implements the OS support for user-level locks, provide significant performance/scalability improvements to multithreaded applications over the earlier 2.4 kernel.

Our experimental suite consists of seven hand-coded microbenchmarks and nine kernels/applications from the SPLASH2 suite. The microbenchmarks show a 10-90% performance improvement in CPU cycles, L2 miss ratios and bus requests for applications that operate on significant amounts of data inside the critical section.
protected by locks. Using Splash2 application suite we also show that some benchmarks exhibit little or no performance improvement.

7.2 Scheduling and Synchronization techniques

In this section, we first present the conceptual overview of our new scheduling policy. Then we provide important details on the Linux 2.6.9 kernel’s scheduler and futex mechanism. Finally, we present the modifications to the linux kernel to implement our policy.

7.2.1 Conceptual Overview

Our OS level synchronization technique relies heavily on the kernel thread scheduler and requires the scheduler to be aware of user-level locks. In other words, the kernel scheduler must be able to identify the thread that currently holds a contended lock. Using this information, the kernel can pin the thread and its execution state to a single cache for reasonable amounts of time. The locality of the critical section data can be exploited by moving threads that require the same lock to the processor that is executing the thread that currently holds the lock. In addition to this, we enable the scheduler to move threads that do not require the lock to other processors in the system, reducing the workload on the processor whose cache holds the lock.

There are number of advantages in doing this. For example, when a lock is released and the thread that blocked on the lock is awakened, it directly uses the data in its local cache rather than doing remote bus requests to fetch the data. This is quite important given the fact that current micro-processors are much faster than the memory subsystem and the system bus, and hence have to use the data in their caches very efficiently.

Additionally this translates to reduction in the number of requests in the bus and hence improvement in the scalability of SMP systems. Further the thread is
able to perform the computation in the critical section faster and release the lock quicker since the data is already present in its local cache. All these advantages should speed up applications that have heavy synchronization overhead.

7.2.2 Linux 2.6. Kernel Scheduler and Synchronization support

The Linux kernel does not differentiate between processes and threads as most operating systems do. In this chapter, any reference to tasks or threads refer to the same entity since we are primarily interested in multithreaded applications. The Linux 2.6 kernel scheduler uses a O(1) algorithm i.e. the scheduler is guaranteed to execute within a certain constant amount of time regardless of the number of tasks currently that are on the system. This is largely made possible by having separate run-queues for each physical processor in the system. The distributed run-queues also enables the scheduler to provide better scheduling support for SMPs.

First, the load on each processor is estimated by multiplying the number of active tasks on the run-queue by a scaling factor. This metric enables the scheduler to handle load imbalance in the system. Additionally, this is also used to enforce better “SMP Affinity” i.e. tasks stay on the same run-queue (or processor) for longer so that they utilize the caches better.

This is a significant improvement over the previous 2.4 kernel where threads randomly migrated across processors resulting in poor performance. However, we propose to migrate tasks when they block for synchronization events since they involve heavy serialization overheads.

The Linux 2.6 kernel scheduler uses a variety of techniques to schedule/migrate threads depending on the system state. One way that threads are scheduled is by the Migration thread which is a per CPU high priority kernel thread. The Migration thread attempts to keep a balance among processors, making sure that one or a
few of the available processors are not carrying a significant portion of the active threads. If the load is out of balance, the Migration Thread will migrate threads from a processor that is carrying a heavy load to a processor or processors that currently have a light load. The migration thread is activated based on a timer interrupt to perform active load balancing or when requested by other parts of the kernel.

Another way that scheduling is done is on a thread by thread basis. When each thread is unblocked or is being scheduled to run, the scheduler will check to see if it can run on its currently assigned processor, or if it needs to be migrated to another processor to keep the load balanced across all processors.

Since the beginning of the Linux 2.5.x kernel series, user level synchronization is supported in the OS by using the kernel *Futex* (Fast User-space muTEX) mechanism. Futexes can be used as building blocks for supporting fast user-space mutexes and semaphores in system libraries since they are lightweight locks. For example, the Linux 2.6 POSIX thread library also called as NPTL (Native Posix Thread Library) uses futexes to implement `pthread_mutex.*` calls.

Operations on futexes start in the user-space but enter the kernel when necessary using the the `sys_futex` system call. For threads that share the same address space, a futex is identified using the virtual address of the lock in the user-space. This address is used to map into a kernel data structure that implements a hashed lock bucket. The mechanism also provides support for inter-process communication, but the details of its working are beyond the scope of this paper.

The Linux man pages defines `sys_futex()` as: “...`sys_futex()` system call provides a method for a program to wait for a value at a given address to change, and a method to wake up anyone waiting on a particular address”. In short the kernel futex mechanism works as follows:
• When a futex is free, a user-level thread can acquire the lock, but it does not need to enter the kernel to do so.

• When a futex is not free (lock is contended), a thread wishing to acquire the lock will enter the kernel. It is then queued along with all other previously blocked threads using the \texttt{sys\_futex(..., FUTEX\_WAIT, ...)} system call.

• When a thread releases a lock that is contended, it will enter the kernel and wake up one or more blocked threads on the corresponding futex using the \texttt{sys\_futex(..., FUTEX\_WAKE, ...)} system call.

7.2.3 Linux 2.6.9 Kernel Modifications

The goal behind our modifications is to create a link between the kernel futex mechanism and the scheduler - in other words, to make the scheduler “futex aware”. This additional knowledge enables the scheduler to make intelligent decisions for threads that are contending for locks. To do this, we modify three parts of the kernel - the thread information table, the futex implementation, and the scheduler.

\textbf{Thread Information Table} This table has an entry called \texttt{CPU} (type unsigned int) to represent the physical CPU on which the task is currently on. We added a new entry called \texttt{cpu\_lock} (type unsigned int) to represent the physical CPU the thread will need to run on when it acquires a lock. Upon the creation of a new thread, this field is initialized to value larger than the number of processors in the system called \texttt{CPU\_LOCK\_INIT}. For example, we initialized \texttt{CPU\_LOCK\_INIT} to 16 but any value greater than the number of processors in the system should work.

\textbf{Futex Mechanism} We added our modifications to the futex mechanism in the \texttt{futex\_wake()} function. When a thread is releasing a futex (or lock), we pass
the value of the `cpu` that it is running on into `cpu_lock` for the thread that it is waking. After doing this, we set `cpu_lock` of the calling thread to `CPU_LOCK_INIT`, as it is no longer waiting for that lock.

**Scheduler** The scheduler checks the `cpu_lock` when migrating a thread or when trying to activate a blocked thread. If the thread is not waking up due to some synchronization event (`cpu_lock=CPU_LOCK_INIT`), then the scheduler performs as usual. In other cases, the modifications to the scheduler drives its decisions based on the following heuristics:

- If a thread is already running on the processor defined by the value in `cpu_lock` value, the thread is not migrated away from that processor.
- If a thread has a `cpu_lock` value defined, migrate this thread to that processor.

Figure 7.1 shows the flow of control within the linux kernel scheduler.

![Diagram](image)

**Figure 7.1. Migration flow within the Scheduler**

The kernel modifications we propose are non-intrusive to the normal operation of the scheduler. Our modifications are not invoked when threads don’t block on contended locks. We allow the scheduler to handle these threads as it normally would. Additionally since only one thread is awakened at a time, the “lock hot”
processor is not running heavy loads. Finally we also do not interfere with *which*
of the blocked threads is awakened but we only change the decision of *where* it is awakened.

7.3 F.A.S.T experimental methodology

This section gives details on the experimental methodology. First, we describe the system under consideration. Then we explain the performance monitoring tools and metrics used in evaluating the effect of our modifications. Finally, we give details on the microbenchmarks and applications that were used to test the system.

7.3.1 Baseline System

The baseline system used for all the experiments is presented in Table 7.3.1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server name</td>
<td>HP Integrity rx4640-8 Server</td>
</tr>
<tr>
<td>SMP Type</td>
<td>4-way Intel®Itanium®@2</td>
</tr>
<tr>
<td>Processor Speed</td>
<td>1.5 GHz</td>
</tr>
<tr>
<td>L1I cache size (line size)</td>
<td>16Kb (64 Bytes)</td>
</tr>
<tr>
<td>L1D cache size (line size)</td>
<td>16Kb (64 Bytes)</td>
</tr>
<tr>
<td>L2 cache size (line size)</td>
<td>256KB (128 Bytes)</td>
</tr>
<tr>
<td>L3 cache size (line size)</td>
<td>4MB (128 Bytes)</td>
</tr>
<tr>
<td>Memory size</td>
<td>8 GB</td>
</tr>
<tr>
<td>Bus bandwidth</td>
<td>12.8 GB/s</td>
</tr>
<tr>
<td>OS kernel (Distribution)</td>
<td>Linux 2.6.9 kernel (Redhat Linux Enterprise)</td>
</tr>
</tbody>
</table>

7.3.2 Performance Monitoring tools and Metrics

We experimented with a number of performance profiling/monitoring tools for the Linux-IA64 platform. These include Pfun [85], Tau [86], and HPCToolKit [87]
to name a few. All these tools use the hardware performance counters to profile applications. Tau and HPCToolkit are more advanced than Pfmon and can be used for finding bottlenecks in the application code. Further these tools also annotate source codes with the performance information and provide a graphical interface for interpreting the results.

For all the analysis in this chapter we use Pfmon. We choose Pfmon primarily because it is simple to use and provides most of the basic metrics required to evaluate performance. We are currently using the other tools to understand the applications better and determine how our policy can work for a wider class of applications.

The Pfmon utility [85] collects counts or samples from unmodified binaries running on IA64 processors (Itanium, Itanium2). It can also be used to profile an entire system. It uses the IA-64 hardware performance counters and the Linux kernel perfmon interface to monitor real applications. It makes full use of the libpfm library to help in programming the IA-64 Performance Monitoring Unit (PMU) [88].

The Pfmon tool gives access to all the features of both Itanium and Itanium2 PMU. For our experiments, we monitored the following hardware events: CPU cycles (CPU_CYCLES), number of bus requests (BUS_ALL_SELF), L2 cache misses (L2_MISSES), L2 cache references (L2_REFERENCES), L3 cache misses (L3_MISSES), L3 cache references (L3_REFERENCES). The L2 and L3 miss ratios were obtained from the ratio of the respective misses and reference numbers. Further, each thread was monitored individually and the metrics we aggregated over the values of all the threads. Finally, we monitored the above events for both user and kernel space.

7.3.3 Application benchmark suite

We compiled the seven lock microbenchmarks (single_ctr, multiple_ctr, doubly_list, prod_cons, affinity_ctr, mlt_lck_mlt_ctr and multiple_fifo) listed in section
3.1. Each thread executes the critical section ranging from 200,000 to 16 million iterations. We also compiled nine of the thirteen SPLASH2 Application/Kernel codes for Linux-IA64. The base problem sizes were used for all the application/kernel codes.

7.4 Results

Each of the application/thread combination was run five times and the results presented in these graphs represent the data averaged over all the runs. For each microbenchmark, we measure the percent improvement for 4,8,16,32,64, and 128 threads. These values were obtained using pfmom with metrics such as CPU_CYCLES, L2_MISSES, L2_REFERENCES, L3_MISSES, L3_REFERENCES, BUS_ALL_SELF and were calculated using the formula ((unmodified kernel value - modified kernel value)*100/unmodified kernel value).

7.4.1 Microbenchmarks results

Figure 7.2 shows the percent improvement in CPU cycles for the seven microbenchmarks in the modified kernel over the unmodified kernel. For example, the affinity_ctr for 128 threads shows an improvement of 50.9% - meaning that the modified kernel ran this program in half the number of CPU cycles than the unmodified kernel. The best improvements were obtained for the 4, 8, and 16-thread cases, with peak improvements as high as 99%.

Overall, an improvement of 18-75% is observed for five of our seven microbenchmarks (affinity_ctr, single_ctr, multiple_ctr, doubly_list, mlt_ctr_mlt_lock) for the range of threads under consideration. The 4-thread case for prod_cons, as well as the 32, 64, and 128-thread cases for multiple_fifo, do not show an improvement, primarily due to the scheduling overhead being higher than the synchronization overhead for these cases.
Figure 7.2. Microbenchmarks: % Reduction in number of CPU cycles

The percent improvement in the L2 and L3 miss ratios is listed in figures 7.3 and 7.4 respectively. All microbenchmarks show an improvement of 6-94% for the L2 miss ratio. This improvement is a key artifact of our modification, translating to a reduction in the number of bus requests, CPU cycles, and L3 cache references. However, it should be noted that the L3 miss ratios do not show these same improvements, with a small negative improvement in many cases. The lower L2 miss ratio generating a lesser number of L3 references in the modified kernel compared to the unmodified kernel. Cold-start misses will still occur, and the combination of these L3 cache misses and the lower number of L3 references result in a higher L3 miss ratio for the modified kernel in many cases.

Figure 7.5 shows the percent improvement in bus requests for the microbenchmarks. As in the CPU cycles case, the 4, 8, and 16-thread cases performed best, with improvements ranging from 10-99%. In general, decreasing the number of threads
Figure 7.3. Microbenchmarks: % improvement in L2 miss ratios

Figure 7.4. Microbenchmarks: % improvement in L3 miss ratios
increased the performance improvement. Again, the 32, 64, and 128-thread cases of multiple_fifo did not show improvement, as the overhead from scheduling this program will outweigh the benefit created from the synchronization improvement.

![Graph showing % Performance Improvement for different benchmarks and thread counts](image)

Figure 7.5. Microbenchmarks: % Reduction in number of bus requests

The affinity_ctr microbenchmark performs best across all metrics, as it is specifically designed to create explicit affinity between the locks and the data accessed within the critical section. In the unmodified kernel, since locks and critical section data are not pinned down to a particular cache, the cache lines get transferred across processors more frequently.

7.4.2 Splash2 Results

The percent improvement in CPU cycles for the nine splash2 applications are shown in Figure 7.6. Our results here, for the most part, fail to form any significant trends, with improvement ranging from a negative improvement of 39% to a positive
Figure 7.6. Splash2: % Reduction in number of CPU cycles

improvement of 10%. We can only provide conjecture as to why some splash2 applications do not run as well as others, and why many do not perform in a consistent basis. The primary reasoning is that these benchmarks were not coded with cache line size in mind, so false sharing and cache conflict misses are occurring often.

Pointers are 64-bit values on Itanium processors, so if an application mixes pointers and scalar data types inconsistently, it will take a performance hit. Also, several of these benchmarks do not spend a high percentage of time performing synchronization, so the scheduling overhead introduced by our modification is not offset by the synchronization overhead. We intend to investigate these issues more thoroughly as part of future work.

Figures 7.7 and 7.8 shows the percent improvement in the L2 and L3 miss ratios for SPLASH2. We see improvements L2 miss ratios ranging from a negative improvement of 5% to a positive improvement of 33%, with most of our cases showing
Figure 7.7. Splash2: % improvement in L2 miss ratios

Figure 7.8. Splash2: % improvement in L3 miss ratios
positive improvement. On the other hand, L3 miss ratios show positive improvement of up to 55%, but also take a higher negative performance hit, as low as 75%. In general, miss ratios improve as the number of threads increase. This is a good sign because with larger number of threads the caches have more conflict misses. However we are able to reduce the effect of conflict misses.

Our bus results vary for each program, but each show a fairly consistent trend of providing better results for a higher number of threads.

![Bar chart showing performance improvement across different benchmarks and thread counts.](image)

Figure 7.9. Splash2: % Reduction in number of bus requests
CHAPTER 8

CONCLUSIONS AND FUTURE WORK

This chapter presents some conclusions and avenues for future research on the work presented in the previous chapters. We have divided the contents based on the three main topics of this work: scalable synchronization mechanisms, active graph based synchronization techniques and finally Futex aware synchronization techniques.

8.1 Scalable Synchronization Mechanisms

We implemented and tested a number of scalable barrier and lock mechanisms for the LWP architecture. The basic idea was to leverage full/empty bits to implement these algorithms. In general, the scalable mechanisms were useful and provided significant performance improvements over conventional spin-wait mechanisms.

We used SST extensively for all our experiments presented in Chapter 5. SST is a very useful tool for performing complex concurrent architectural simulations involving large number of threads and architectural components. Its ability to simulate both conventional and LWP specific features was useful in performing comparative studies across these architectures. Most importantly, the support for compiling and simulating legacy codes such as SPLASH2 and NAS was important to really understand the usefulness of the proposed mechanisms.
For the most part we were able to easily make some small yet important modifications to SST such as thread scheduling and adding new system calls. SST is definitely a tool that we should use in future for our experiments. We list some possible enhancements to SST in the below.

Future work for scalable synchronization mechanisms will need to look at some of the following issues:

**New barrier and lock algorithms:** A number of scalable synchronization mechanisms have already been proposed for different architectures in the past. In this thesis we implemented only a subset of those mechanisms for the LWP architecture. However, it should be an interesting exercise to implement other mechanisms that may have different behaviors than the ones presented. More importantly, we need to explore novel LWP specific algorithms that are most suited for light weight threads.

**SST (Synchronization Trap Handler):** SST currently queues/wakes up all the threads which block on a single memory location using a hardware mechanism. However, implementing this completely in hardware may not be cost-effective. It should be more realistic to have a combination of hardware and software mechanisms to take care of the multiple producer/consumer problem. From a software perspective, we need to be able to spawn a trap handler when the hardware encounters multiple threads contending for a single memory location.

**SST (EMS):** Currently SST supports only two memory states for the full/empty bits, namely full and empty. This is a simplification over the original proposal which supports eight to ten extended memory states. It should be interesting to provide support for all these states and allow threads to interact with these additional memory states.
**SST (threads):** A number of possible enhancements to SST will make it more useful for investigating large scale LWP based architectures. Currently, threads in SST have a stack and additional baggage as most conventional implementations of threads do. It would be interesting to support a more classical definition of light weight multithreading, where the thread’s state consists of only a register frame. This should be done transparently to the programming model to allow us to simulate legacy codes.

**SST (thread frames):** Currently thread frames in SST are not part of the address space. This is necessary to capture thread-to-thread communication using thread frames.

**SST (components):** It would also be important in the long run to add new architectural components and more importantly to simulate the existing components with more architecture details. These may include: on-chip interconnection network for LPC/CMP, LWP pipeline, LWP thread scheduler, cluster of SMPs and accurate delays for performing different local and remote memory operations, to name a few.

**SST (tools):** SST currently does not have a debugger or a visualization tool. In future, these tools are important for performing and interpreting large scale simulations. Currently, users have to debug SST using GDB rather than debug the application running on SST. This is a hassle and makes it difficult for fixing the simulator or identifying bugs in the application.

8.2 Active Graphs based synchronization

Chapter 6 presented some active graph based synchronization mechanisms. We experimented with three barrier algorithms based on active graphs. These barriers
leverage thread-to-thread communication, full/empty bits, and light weight thread concepts to exploit parallelism and improve performance of barrier mechanisms.

All our experiments were conducted using the DimC/SALT toolset. The DimC language and compiler enabled us to write multithreaded code specifically targeted for the LWP architecture. The SALT simulator allowed us to simulate the code generated by DimC. DimC/SALT provides support for memory mapped thread frames, extended memory states, light-weight threads and software trap handler etc. All these allowed us to capture the idea of active graphs with this framework.

We observed that as we increase the number of threads, the contention for a single memory location and the overheads of the EMS handler thread increases. This clearly showed the need for scalable synchronization mechanisms for the LWP architecture.

In general, the dissemination barrier outperformed all the other barriers based on cycle time and total instructions committed. It requires 7 – 10% less cycles than the tree and the tournament barrier. In short, the overheads of the EMS handler is much larger than spawning off additional barrier threads and making them perform the barrier algorithm.

Future work in this area will explore some of the following issues:

**Active graphs:** These have a great potential for solving large scale graph problems. It would be interesting to build additional lock and barrier algorithms based on active graphs.

**Active graph synchronization:** Due to the limitations in the toolkit, we were able to obtain minimal statistics on the performance of active graph based synchronization mechanisms. In future, more extensive analysis is necessary for understanding these mechanisms.
**Language extensions:** The DimC compiler was a dedicated compiler but lacked a runtime library and other features for running legacy code. It should be interesting to modify an existing C language compiler to include subset of these features. A good starting point would be a re-targetable compiler such as LCC [89].

**Simulator:** As mentioned above, we plan to add features to SST that would allow us to perform all these experiments under one set of tools. These include features such as EMS, software handler, and memory mapped thread frames to name a few.

8.3 F.A.S.T

The Linux kernel modification presented in Chapter 7 provided significant performance gains for microbenchmarks. By exploiting the locality of the critical section data by enforcing an affinity between locks and the processor, we have shown the possibility of large performance gains. We also showed that when an application is properly written with cache line size in mind, we can further decrease the L2 miss ratio, and in turn lower the L3 miss ratio, the number of bus requests, and the CPU cycles for an application.

However, SPLASH2 applications produced mixed results. In general, the scheduler modifications helps in improving the performance for highly multithreaded scientific applications.

Future work in this area will explore some of the following issues:

**Application profiling:** We plan to extend our testing to a wide variety of applications and testing suites, so we can gain a better understanding of situations our modifications improve the overall performance of a system. Further, this is necessary to understand the limitations of our approach. We are currently
using tools such as HPCToolkit and Tau to profile our applications to identify bottlenecks in our approach. Specifically, this would allow us to understand the behavior of SPLASH2 benchmarks.

**O(1) scheduler guarantees:** We are continuing our efforts to optimize the scheduler by obtaining execution traces of the scheduler in order to fine-tune it and limit the overhead introduced by our modifications. We are using some kernel tracing tools [90] to track events within the kernel. Part of this effort is to ensure that our modifications does not break the O(1) scheduling guarantees provided by the current Linux schedulers.

**Adaptive strategies:** Our modification may create a fair share of scheduling overhead for some applications that do not perform a lot of fine grain synchronization. Once we have identified such situations, it would be beneficial to have an adaptive scheduling strategy to migrate threads only when necessary. The scheduler would default to existing scheduling techniques when the cost of thread migration is too large.

**Multi-core processors:** We would also like to test our scheduling technique on multi-core SMP systems. We hypothesize that our modifications could allow for even larger improvements because it is necessary to use the caches more efficiently in such systems.

8.4 New Directions

This work can also be extended in a number of new and interesting areas.

8.4.1 Analytical models

Analytical models are one of the most important mathematical tools used by architects to understand the tradeoffs involved in various aspects of an architecture.
In most cases analytical models can coexist with corresponding architectural simulators and can direct the simulation towards the most useful points in the design space. Analytical models also benefit from simulators as the latter is usually used to verify and fine-tune the former for well known design points. Most importantly, analytical models can be used to predict system behavior for configurations where simulations may be too expensive in terms of time and system resources required.

Analytical models do have some limitations. The primary limitation is the ability to represent complex application and (micro-)architectural features as mathematical concepts. Further in novel architectures the designer should be aware of the most important aspects of the architecture that need to be represented in the model, but in most cases this information is not usually well defined until some mathematical or simulation data is available. Therefore, most analytical models are kept as simple as possible and hence ignoring many underlying idiosyncrasies of the system under consideration.

A number of analytical models have been developed for parallel architectures [91][92], multithreaded processors [93] and synchronization mechanisms [37][38]. Most of these are probabilistic models and make a number of assumptions that may not be realistic. We are interested in a special class of analytical models based on queueing theory called Mean Value Analysis (MVA) models. MVA models have been specifically popular with the architecture community for modeling [94][95][96].

Recently, MVA models were used to model shared memory architectures with ILP processors [97][98] and some synchronization overheads [99] involved in such architectures. We plan to model the LWP architecture and synchronization mechanisms based on MVA models and compared it against the simulations presented in this chapter. This would enable to understand very large-scale LWP systems with millions of threads and processors.
8.4.2 Transactional memory

Lock-based programming has a number semantic and performance disadvantages. Locks are prone to semantic problems such as deadlocks, convoys and priority inversion etc. Locks also impose unnecessary constraints that affects performance. A simple example would be a large array protected by a lock. This ensures all accesses to the array is serialized. However, this results in simplicity of implementation at the cost of concurrency. Even if multiple threads update different array elements, each update is serialized. Using more locks to protect the array does not solve the problem either since it requires additional programming effort and may lead to deadlocks if locks are misused.

One of the techniques that has recently obtained a lot of attention with the architecture community is the idea of Transactional Memory (TM). TM is a concurrency control mechanism for enabling concurrent access to shared memory by multiple threads. This is closely related concept of database transactions [100]. Both these techniques guarantee Atomicity, Consistency, Isolation and Durability (ACID) properties for ensuring correctness.

The underlying TM implementation ensures these properties for segments of code that need to execute atomically. In other words, TM serializes concurrent accesses only when they conflict unlike locks which serializes all operations to shared data. Depending on the system, the code segments may be identified by the programmer, compiler or automatically by the hardware.

Transactional memory can be classified into Hardware Transactional Memory (HTM) [57][55][101][102] and Software Transactional Memory (STM) [103]. A number of hybrid approaches have also been proposed recently [104].

HTM proposes a number of enhancements to existing processor architecture to support transactional memory execution in hardware. These mechanisms have been
traditionally dependent on cache coherency protocols to detect conflicts and commit
the changes atomically. HTM mechanisms do have a number of software limitations
such as context switches and size/durations of transactions that sometimes over-
shadow the performance advantages of an hardware implementation.

Software Transactional Memory (STM) relies on languages, compilers and li-
braries to provide support for TM mechanisms. STM uses concepts of non-blocking
synchronization and lock-free execution to enable concurrent access to shared ob-
jects. Early STM mechanisms had high overheads since transactions had to be
managed completely in software. However, the more recent proposals [105] have
comparable overheads. STM does not have the problems of HTM implementations
on size/duration of transactions but nevertheless introduce new programming prac-
tices that takes longer to adapt.

The challenge here is to develop a light weight TM mechanism for the LWP
architecture. We currently foresee a hybrid mechanism that provides a low-overhead
approach for implementing TM for a large number of threads.

8.4.3 UPC on SST

One of the strengths of SST is in its ability to run legacy codes on the LWP
architecture. This has been primarily possible because of the successful adaption of
programming models such as OpenMP and MPI. In recent years, UPC has become
increasingly popular with the high performance computing community. Adapting
UPC would be useful in expressing parallelism and exploring a different kind of
programming model for the LWP architecture. UPC combines the advantages of
both shared address space abstraction with that of message passing. UPC allows
the programmer to combine the ease of programming of shared address space with
the data layout capabilities and performance of message passing paradigm.

149
UPC [106] is an explicit parallel extension of ANSI C and supports a distributed shared memory programming model. UPC extends ISO C 99 with the following constructs:

- An explicitly parallel execution model
- A shared address space
- Synchronization primitives and a memory consistency model
- Memory management primitives

There are some significant challenges in adapting UPC to the LWP architecture. UPC is based on Single Program Multiple Data (SPMD) model of computation where the amount of parallelism is fixed and specified at program startup time. However, one of the strengths of LWP architecture is its ability to create lightweight threads on the fly at low cost. This and other issues make this an interesting area of research.
BIBLIOGRAPHY


