EFFICIENT USER-LEVEL EVENT NOTIFICATION

by

Michael Allen Parker

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2013

Copyright © Michael Allen Parker 2013

All Rights Reserved

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Michael Allen Parker**

has been approved by the following supervisory committee members:

Alan Davis	, Chair	**3/18/2013** Date Approved
John Carter	, Member	**3/18/2013** Date Approved
Wilson Hseih	, Member	**3/18/2013** Date Approved
Ganesh Gopalakrishnan	, Member	**3/18/2013** Date Approved
Rajeev Balasubramonian	, Member	**3/18/2013** Date Approved

and by **Alan Davis** , Chair of

the School of **Computing**

and by Donna M. White, Interim Dean of The Graduate School.

ABSTRACT

High-performance supercomputers on the Top500 list are commonly designed around commodity CPUs. Most of the codes executed on these machines are message-passing codes using the message-passing toolkit (MPI). Thus it makes sense to look at these machines from a holistic systems architecture perspective and consider optimizations to commodity processors that make them more efficient in message-passing architectures.

Described herein is a new User-Level Notification (ULN) architecture that significantly improves message-passing performance. The architecture integrates a simultaneous multithreaded (SMT) processor with a user-level network interface (NI) that can directly control the execution scheduling of threads on the processor. By allowing the network interface to control the execution of message handling code at the user level, the operating system (OS) related overhead for handling interrupts and user code dispatch related to notifications is eliminated. By using an SMT processor, message handling can be performed in one thread concurrent to user computation in other threads, thus most of the overhead of executing message handlers can be hidden.

This dissertation presents measurements showing the OS overheads related to message-passing are significant in modern architectures and describes a new architecture that significantly reduces these overheads. On a communication-intensive real-world application, the ULN architecture provides a 50.9% performance improvement over a more tradi-

tional OS-based NIC and a 5.29-31.9% improvement over a best-of-class user-level NIC due to the user-level notifications.

Dedicated to my best friend and son, Kevin Michael Parker

TABLE OF CONTENTS

ABSTRACT iii

LIST OF FIGURES ix

LIST OF TABLES xiii

ACKNOWLEDGEMENTS xiv

1 INTRODUCTION 1

2 MESSAGE PASSING COSTS 10

2.1 Message Latency and Overhead Models 10
2.2 Understanding Message Overheads 12

3 ULN ARCHITECTURE 19

3.1 The SMT Processor 19
3.2 User-level Notification and Synchronization Table 21
3.3 User-level Network Interface 22
3.3.1 Message Send Mechanisms 23
3.3.2 Message Arrival Mechanisms 24
3.4 Messaging Protocol 26
3.4.1 Message Descriptors 27
3.4.2 Message and Packet Formats 29
3.4.3 Connection Setup and Protection 31
3.5 Network Model 31
3.6 The Journey of a Message 33
3.7 Summary 34

4 MICROBENCHMARKS 36

4.1 Simulator 37
4.2 Messaging Overheads 38
4.3 Ping-Pong 41
4.3.1 Polling-Based Ping-Pong 43
4.3.2 Notification-Based Ping-Pong 45
4.3.3 Receive Thread Ping-Pong 47

4.4 SMT Performance ... 51
4.5 Polling versus User-Level Notification Overheads ... 53
4.6 TAU Measurement Overhead ... 55
4.7 Manual Instrumentation Overhead ... 57
4.8 Speedup of Modern Core ... 58
4.9 Microbenchmark Summary ... 59

5 REAL-WORLD APPLICATION: DIRT ... 61

5.1 The Real-Time Ray-Tracer ... 61
5.2 The DIRT Derivative ... 63
5.3 DIRT Details ... 65
5.3.1 Node Initialization ... 66
5.3.2 Worker Node Operation ... 66
5.3.3 Software Shared Memory System ... 67
5.4 Key Points ... 68

6 DIRT CHARACTERIZATION ... 71

6.1 Experimental Setup ... 71
6.2 Renderer Thread Characterization ... 75
6.2.1 Renderer::run Function ... 75
6.2.2 popTask Function ... 77
6.2.3 traceRay Function ... 78
6.2.4 intersect Function ... 78
6.2.5 isect Function ... 79
6.2.6 getrhos_many Function ... 79
6.2.7 get_data Function ... 81
6.2.8 Renderer Thread Summary ... 84
6.2.9 Renderer Thread Analysis ... 84
6.3 Communicator Thread Characterization ... 89
6.3.1 communicator::run Function ... 89
6.3.2 dataserver_group::handlemessage Function ... 91
6.3.3 dataserver_direct::handlemessage Function ... 91
6.3.4 communicator Summary ... 92
6.3.5 32w_1r Communicator Thread Analysis ... 94
6.3.6 Combined 32w_1r Analysis ... 94
6.4 All DIRT Runs ... 95
6.5 Cache Miss Model ... 100
6.6 Characterization Summary ... 103

7 A MATHEMATICAL MODEL OF DIRT ... 105

7.1 Renderer Thread Model ... 105
7.2 Communicator Thread Model ... 109
7.3 Expected Message Queueing Latency ... 110

7.4 Multiple Thread Performance Model 112
7.5 Load Imbalance 117
7.6 Combined Model 123

8 DIRT ON ULN 124

8.1 SMT Performance 124
8.2 Scalability 128
8.3 Wire Latency 129
8.4 Send and Receive Overheads 131
8.5 Thread Communications 135
8.6 DMA versus PIO 138
8.7 ULN Performance Summary 139
8.8 What-if: Finer-grained Cache Block 140
8.9 What-if: Directly Resume Renderer 143
8.10 Combined Benefits 147

9 RELATED WORK 149

9.1 Simultaneous Multithreading 149
9.2 Thread Synchronization 150
9.3 Efficient Protocols 151
9.4 Message Passing Architectures 154
9.5 Tighter NIC Integration 161

10 CONCLUSIONS AND FUTURE WORK 162

REFERENCES 167

LIST OF FIGURES

1 Anatomy of a message for a kernel-mode NI 15

2 Anatomy of a message for a user-level NI without user-level notifications 15

3 High-level ULN node architecture block diagram 20

4 Packet data injection flow-chart 25

5 Message descriptor structure 28

6 Message structure 30

7 Anatomy of a message in the ULN architecture 34

8 Ping-pong microbenchmark: main() 43

9 Ping-pong microbenchmark: sync() 44

10 Poll-based ping-pong: nodeA_side() 45

11 Poll-based ping-pong: nodeB_side() 46

12 ULN-based ping-pong: nodeA_side() 47

13 ULN-based ping-pong: nodeB_side() 48

14 Multithreaded notification-based ping-pong: nodeA_side() 49

15 Multithreaded notification-based ping-pong: communications thread 50

16 Multithreaded notification-based ping-pong: nodeB_side() 50

17 Measured SMT speedup versus best-fit model 53

18 TAU instrumentation overhead measurement technique 55

19 Manual instrumentation counter macros 57

20 Manual instrumentation timing macros 58

21 Sample images from DIRT run 72

22 Annotated Renderer thread run function (main loop) 76

23 Annotated Renderer thread popTask function 77

24 Annotated Renderer thread traceRay function 78

25 Annotated Renderer thread intersect function 79

26 Annotated Renderer thread isect function 80

27 Annotated Renderer thread getrhos_many function 81

28 Annotated Renderer thread get_data function 82

29 Renderer compute components 86

30 Renderer communication and load balance components 86

31 Renderer thread breakdown summary 87

32 Corrected Renderer thread breakdown summary 88

33 Annotated communicator thread main function 90

34 Annotated communicator thread dataserver_group:: handlemessage function 91

35 Annotated communicator thread dataserver_direct:: handlemessage function 92

36 communicator thread breakdown summary 94

37 Combined breakdown summary 95

38 Corrected combined breakdown summary 96

39 Total render time versus number of nodes 97

40 Relative speedupversus number of nodes 97

41 Total render time versus number of Renderer threads 98

42 Relative speedup versus number of Renderer threads 99

43 Total time summary versus node count for all runs 99

44 Total time summary versus thread count for all runs 100
45 Combined time summary versus node count for all runs 101
46 Combined time summary versus thread count for all runs 101
47 Cache miss model fit..102
48 Markov model of isomorphic threads on an SMT processor..................................113
49 Markov model of DIRT threads on an SMT processor 114
50 Value of In for n=1 to n=128 122
51 Value of In for n=1 to n=10,000 122
52 Per-thread speedup as a function of thread count 125
53 SMT speedup at multiple nodes...126
54 Markov model state probabilities 127
55 Per-thread speedup as a function of thread count 128
56 Scaling to 512 threads..129
57 Scaling efficiency 130
58 Slowdown on 32 nodes due to wire latency for Nr = 1 to 7 131
59 Slowdown on 32 nodes due to send overhead for Nr = 1 to 7................................132
60 Slowdown on 32 nodes due to receive overhead for Nr = 1 to 7............................133
61 Slowdown due to 1-2 microsecond notification overhead, Nr = 3 134
62 Speedup of ULN versus OS-based NIC, Nr = 3 135
63 Speedup of ULN thread wake versus OS-based lock, Nr = 3, 32 nodes 136
64 Impact of additional latency and overheads...140
65 Optimal cacheline size, expected cache-miss behavior 142
66 Optimal cacheline size, worst-case cache-miss behavior 143

67 Comparison of worst versus expected versus best case 144

68 Speedup of optimized response returns versus best-case block without 146

69 Speedup of optimized code over best-case 7x7x7 block 147

70 Speedup of ULN plus what-if optimizations, $N_r = 3$ 148

LIST OF TABLES

1 Polling message timing 39

2 Notification-based message timing 41

3 Message-passing latencies and overheads 42

4 SMT thread speedups 52

5 Summary of key speedups and overheads 60

6 Summary of Renderer thread key function times across all nodes 85

7 Summary of key Renderer thread characteristics 88

8 Summary of communicator thread key function times across all nodes 93

9 Summary of key communicator thread characteristics 96

10 Summary of key DIRT characteristics 104

ACKNOWLEDGEMENTS

I received an incredible amount of help, support, and encouragement from family, friends, coworkers, and faculty throughout the process of pursuing this degree. I would be negligent not to express just a tiny portion of that gratitude here.

Kevin, thank you most for being a great son and friend. Thank you for inspiring me and even helping me on occasion in this effort. Katie, thank you for reviewing my dissertation and serving as a technical editor all these years. Cinnamon (Zig), thank you for all of your love and support throughout the process, you are greatly missed.

I would also like to thank my older brother, Dr. Steven G. Parker, and my father, Dr. Gregory A. Parker who encouraged me as I pursued my PhD and provided feedback and significant guidance along the way. Thank you both for the countless hours you spent discussing ideas, giving me feedback, and helping me reason about the models in here. Thank you, Steve, for helping me understand DIRT and raytracing in general. You have inspired me as you have pushed forward and helped me even in some tough periods of your own life. Dad, thank you especially for the help with the calculus in the load-balancing model. Thank you both for all that you have done and all you do.

I would also like to thank my mother, Jeanene Parker, PA. She has encouraged and inspired me throughout the process. Thank you, mom. Your love and support over the years has made me who I am today. I have also received a great amount of encouragement and

love from my other brothers and sisters, Sheryl, Jennifer, Tamara, Marilyn, William, and Christopher. Thank you each for all that you have done and continue to do.

My entire committee has been quite helpful and encouraging. I would like to thank Dr. Wilson Hsieh and Dr. John Carter for their extensive guidance and mentoring. Wilson, thank you for your guidance and patience especially in my early graduate career. John thank you for mentoring and employing me for several years. Dr. Rajeev Balasubramonian and Dr. Ganesh Gopalakrishnan have also given me a lot of support. To my whole committee, I appreciate your support and encouragement to finish even long after I left for industry. And last, but definitely not least, I am so grateful to Dr. Al Davis, my advisor, for sticking with me through the process and for encouraging me not to give up in some of my more desperate hours when it seemed like there was no way to make this degree a reality. Both you and Julie have been great friends over the years!

I would like to thank Dr. Neil Cotter, a professor in the Electrical and Computer Engineering Department at the University of Utah, for pointers on extreme value theory used to produce my load balance model. And thank you to countless other faculty and staff in the School of Computing. You have all been great! Thank you, Dr. Erik Brunvand for being such a great friend all these years.

Many friends have supported and encourage me along the way, far too many to list. Thanks to Dr. Lambert Schaelicke, Dr. Binu Mathew, and Dr. Ali Ibrahim who spent many hours giving me feedback, advice, and encouraged me. I would also like to thank Kurtis Bleeker, for countless hours of encouragement, advice and just listening. To my many friends at NVIDIA and Cray that have supported me as well, I thank you all. Especially Dr.

Steve Scott, Dr. Abdulla Bataineh, Greg Faanes, Dr. Zvika Guz, and Dr. Steve Keckler, thanks for encouraging me to complete my degree.

This work was sponsored in part by DARPA and AFRL under agreement number F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official polices or endorsements, either express or implied, of DARPA, AFRL, or the U.S. Government.

CHAPTER 1

INTRODUCTION

Clusters based on commodity processors and interconnects are currently the most common platform for high-end supercomputers. Of the machines listed on the November 2012 Top 500 Supercomputer Sites list [134], 413 or 82.6% are reported to have Gigabit Ethernet or Infiniband as their primary interconnect. An additional 3 machines are based on Myrinet. Furthermore, 440 or 88.0% of these machines are based on the commodity x86 microprocessor [116,125], with an additional 58 based on commodity SPARC [113] or Power [126] processors. Only two machines report custom processors. Virtually all of the machines are considered massively-parallel processor architectures (MPPs) or clusters.

In November 2005, 392 or 78.4% of the top 500 machines were based on an Infiniband, Ethernet, Quadrics or Myrinet cluster interconnect. MPPs and clusters made up 464 of the 500 machines. The processor family for 343 of these machines is based on the x86 processor, with another 143 listed as commodity Sun SPARC, Intel IA-64, IBM Power, Digital Equiptment Alpha [117], or Hewlett-Packard PA-RISC [17] processors. In November 2000, only 33 were cited to be based on such a cluster interconnect, with 274 reported as MPPs or clusters. Additionally, 450 were based on commodity processors, only 6 of which were x86. These machines are increasingly becoming based on commodity interconnects and processors.

The most common parallel programming model used on cluster and MPP systems is message-passing, using the Message-Passing Interface (MPI) [130]. Given the popularity and prevalence of commodity clusters and commodity-processor-based MPP systems, there is reason to consider optimizations to commodity processors that make them more efficient in message-passing architectures.

Current trends in very large-scale integrated (VLSI) chip design [19,80,127] are that transistor counts on a fixed-size die are quadrupling approximately every 3 to 4 years. In the past 7 years, we have seen the number of commodity x86 central-processing unit (CPU) cores in a socket or on a single die grow from a single core in early 2005 to dual CPU configurations with the dual-core AMD Athlon die and dual-die Intel Pentium D by the end of 2005 to the 10-core single-die Intel Xeon E7 and the 16-core dual-die Opteron 6200 in early 2012. In the commercial research domain, Intel has addressed higher core counts with the 80-core Tera-scale research platform and the 48-core Single-Chip Cloud Computer. The Intel Many Integrated Core (MIC) architecture consists of in excess of 50 simpler x86 cores on a single chip. Other commercial high-core count processors include the Tilera TILE, TILE-Pro, and Tile-Gx [133], the NVIDIA Tesla [74], the AMD Radeon [116], and the Clearspeed CSX700 [118]. While each vendor appears to count cores differently, each of these processors contains a high degree of thread resource and function unit replication.

As on-die concurrency increases through both an increase in core count and an increase in the number of functional units per core, the demand for bandwidth into and out of the processor chip increases [96]. High-capacitance multidrop processor interconnection and input/output (I/O) buses have not kept pace with this demand. Their failings are many.

The bandwidth on a multidrop bus is shared among all of the devices attached to the bus. Electrical transmission line effects such as the capacitance of multiple devices on a bus and reflections caused by impedance discontinuities from stubs in the multidrop bus architecture severely limit the achievable bandwidth, length, and number devices on these buses.

Due to these limitations, I/O and processor interconnect interfaces are increasingly becoming point-to-point interfaces that look much more like message-passing networks than the traditional broadcast multidrop buses. Legacy multidrop I/O buses such as PCI [132] have been replaced by point-to-point interconnects such as PCI-Express [131], USB [135], FireWire [120], HyperTransport [121], and InfiniBand [122]. In addition, processor front-side buses have been replaced by point-to-point interconnects such as AMD's HyperTransport-based Direct Connect Architecture [121] and Intel's QuickPath Interconnect (QPI) [123]. In the memory system, we are also starting to see the traditional multidrop stub-bus architecture reach fundamental limits [28, 55]. Candidates for point-to-point memory interfaces such as Fully-Buffered Dual Inline Memory Module (FBDIMM) [119], buffer-on-board approaches [124, 128], and Rambus dynamic random access memory (RDRAM) [34] are current evidence of this trend.

To this end, it is important to work toward an architecture that can be efficiently interconnected both on- and off-chip. Connections on- and off-chip should be point to point. Communication over these point-to-point networks can be viewed as low-level message-passing, where queries are sent to devices and responses are subsequently received. Since technology trends force the hardware to use point-to-point links, there is an interesting opportunity to expose communication directly to user-level software through a message-passing interface. By looking at this opportunity from a holistic systems point of view from

user-level software down to hardware, we can dramatically reduce the latency and overhead for both processor-to-processor and processor-to-I/O message-passing.

Previous research has shown that latency and overhead in message-passing systems limit parallel efficiency and scalability [79]. As a result, considerable effort has been focused on reducing overheads in message sends and receives. Proposals to virtualize network interfaces to reduce OS involvement in messaging have shown that the benefit of giving user-level code direct access to efficient network interfaces can be significant [12,41,44,50,86]. User-level access to the network interface combined with pushing much of the send and receive overhead onto the network interface (NI) reduces the software overhead associated with copying message data onto the NI, triggering the sends, and copying received messages back into main memory.

Message notification is defined as the process by which the user code is informed that a message has arrived. As the overheads of sending and receiving message data have been reduced, the overhead of message notification has become one of the more significant contributors to message overhead and latency. It is the limiting factor in many programs [79]. This is particularly true for codes that need to frequently communicate small or medium sized messages where the approximate arrival times of messages at a particular receiver cannot be statically determined. As a result, a large degree of effort is often required to design codes to use larger messages in concert with a predictable communication pattern. Recoding in this way not only results in more time and effort spent in code development, but often contributes to code complexity and results in lower run-time efficiency on other parts of the code. Rather than simply living with a message-passing architecture that requires message aggregation, it makes sense to design an interface that allows for simple and

efficient fine-grain communication, thus allowing the programmer to embed communications at the places they naturally occur in codes.

The first step in accomplishing this goal is to understand the problems with message arrival notification in existing architectures. Message arrival notification has traditionally been accomplished either by polling for arrived messages or by posting an interrupt when a message arrives. Each of these mechanisms has significant overhead.

Polling, or periodically checking for a message arrival, has three primary disadvantages. First, the polling code must be inserted into the message-passing program. This often requires the program writer to call a polling routine periodically. Not only does this break up the flow of the program, but knowing when and where to insert such calls can be non-trivial. In the case of MPI programs, checks for message arrival are only being performed during the execution of an MPI library call. When the main user-code is being executed, no polling occurs. Second, the act of polling consumes CPU cycles and power, and adds overhead to the program when no message is there. Third, there is a trade-off between overhead and effective latency. Frequent polling results in low effective message latency but higher polling overhead. Conversely, infrequent polling results in less overhead, but significantly increases the effective message latency perceived by the user process.

Most of the effort in reducing the costs of message notification has been in the context of special-purpose space-shared batch-scheduled systems. In these systems, polling may be acceptable from a system performance perspective when compute and communication patterns are coarse-grained and regular. Such a code may compute for some time, stop and exchange information between processors in the job, and then resume computation. Additionally, in a system without direct-attached I/O or other sources of random interrupts,

a lightweight kernel can fairly quickly deliver an interrupt to the only running user thread. However, even in these special purpose systems there is unnecessary operating system (OS) overhead and there is an energy impact of polling.

The overhead of triggering an interrupt is that the operating system must be invoked to handle a user-level task. The overheads associated with these mechanisms for message notification are discussed in more detail in the following chapter. In general-purpose time-shared multiuser systems the overheads induced by the operating system when an interrupt occurs can be significant. A big part of the inefficiency of processing interrupt-based notifications is due to the legacy view that interrupts are expected to be infrequent. In a fine-grained message-passing environment that uses interrupts for notifications, this is not the case. In a multicore chip, with messaging between cores, such as in the Intel Single-Chip Cloud (SCC), the cost of invoking the OS to handle a message may mean going off-chip to fetch OS code and data just to handle a message from a neighboring core.

This dissertation shows that hardware and software support for user-level messaging, including user-level notifications improves performance for fine-grained message-passing. One of the main contributions of this work is to provide a mechanism whereby the network interface can directly deliver notifications to a user-level process without the aid of the operating system. This is accomplished by allowing the NI to share some control over thread execution. The User-Level Notification (ULN) message-passing architecture presented herein addresses general message-passing latency and overhead problems from a full-system perspective, with a key goal of tackling the message arrival notification overhead problem. This is done with a combination of the following ideas:

- A Simultaneous MultiThreaded (SMT) processor allows the overhead of message handlers to be effectively hidden.
- By convention, using of one or more threads on the SMT processor as communications threads with the remainder used as computation threads.
- A network interface that supports user-level access can be tightly coupled to the CPU to avoid the overhead and latency of slower I/O buses.
- An event synchronization table through which communication threads can park and wait for notifications from the NI, thus eliminating OS involvement in the common case.
- Messages are sent out of and placed directly into the processor cache, reducing message latency and memory bandwidth.
- A zero-copy message protocol that allows messages to be delivered directly to user-space without copying.

Not all of these ideas are new. For example, previous research has explored user-level network interfaces [12,41,44,50,86] and efficient protocols [2,9,20,42,76,100,107, 115]. However, this specific combination of features is both unique and synergistic. Exposing notification mechanisms directly to user-level programs is unique. The event synchronization table, though built upon previous ideas for communication between threads within a single processor, is new in the ULN architecture. The important aspect of this architecture lies in its support for user-level messaging, in a general-purpose commodity cluster system running an off-the-shelf operating system with small modifications to an SMT processor. The key to the approach is to consider the full system rather than to focus on a single aspect of the message-passing system.

The combination of features in the ULN architecture reduces message handling overheads dramatically, without requiring gang-scheduling or forcing a change to the message-notification model seen by the user-level software. The SMT processor, originally targeted to hide memory latency, makes it possible to overlap computation and communication without the complexity and overhead of a secondary communication processor. The combination of a zero-copy protocol and user-level access to the network interface allows user code to communicate without the overhead of OS involvement or excess data copying. Finally, integration of the NI with the SMT processor allows the NI to communicate message arrival events back to the target thread without most of the overhead that an interrupt-style notification would incur.

The architecture presented in this work provides an event synchronization table whereby the network interface can directly deliver notifications to a user-level process, without the aid of the operating system. By promoting the NI to a coprocessor level, like a floating-point unit, direct memory access (DMA) engine, or application accelerator, the NI and CPU can interact more efficiently. This is done by allowing the NI to deliver notifications back to the process via shared control over thread execution.

This dissertation shows that an SMT processor and direct user-level access to hardware messaging mechanisms can reduce and hide communication latency and overhead. Multiple concurrent threads can be used to hide overhead by allowing send, receive, and notification processing to proceed in parallel with normal program execution. However, the more significant contribution of this work is that it shows the benefit of a user-level notification mechanism to reduce overheads associated with message arrival. A hardware lock mechanism can be used to allow the NI to directly deliver message arrival events to a user

process, without involving the operating system. Eliminating kernel overhead in notifications not only significantly reduces message-passing overhead, but also reduces, by tens of microseconds, the end-to-end latency that the user context perceives.

Chapter 2 characterizes the overheads of interrupts in modern processors. Chapter 3 presents the ULN architecture. Chapter 4 shows key characteristics of the ULN architecture which are evaluated through a combination of simulation and analysis. Chapter 5 and Chapter 6 discuss a real-world compute-intensive application and its characterization, respectively. Chapter 7 describes a mathematical model of that application on the ULN architecture. Chapter 8 presents analysis of ULN based on that model and characterization. Chapter 9 discusses related work. Finally, Chapter 10 summarizes this work and suggests future research directions to further this work.

CHAPTER 2

MESSAGE PASSING COSTS

As there are many differing definitions of key message-passing characteristics, such as latency and overhead, it is important to clarify the meanings of certain terms. Many models have been proposed as ways to parameterize key characteristics of message-passing codes and architectures. Common models include Hockney [51], LogP [27], LogGP [8], and PLogP [61]. Further analysis of these models is presented by Pjesivac-Grbovic [95].

2.1 Message Latency and Overhead Models

The Hockney model is a simple straightforward model that estimates communication latency as the sum of the time it takes to transfer a minimal-sized zero-payload message plus the number of payload bytes in the message divided by the effective interconnection bandwidth. This model does not further break down the components of message latency or bandwidth.

The LogP model further breaks down the components of message passing into a latency component (L), an overhead component (o), an intermessage gap (g), and a processor count (P). Latency in this model is defined as the network transit time, or the time from the first byte entering the network interface on the source processor node until the time the first byte comes out of the network interface at the destination. It does not include time spent in the send or receive calls required to process the message. The overhead includes the time

the processor is occupied on either end to process the send or receive. The gap is the minimum interval between consecutive message sends or receives for minimal sized messages. This gap is usually an artifact of the overhead required to send a message and limits the message throughput for small messages. The primary source of this overhead is primarily the time spent processing the send call and the time spent interacting with the network interface to initiate the send. The message latency that a user observes in this model is the sum of the message traversal latency and the sender and receiver overheads.

The LogGP model builds upon the LogP model by adding an additional gap parameter (G), which is the effective bandwidth of the system on long messages. This parameter distinguishes between the software message initiation overhead and the effective network bandwidth in most systems. In the LogGP model, the message latency that a user observes in a real system is sum of the wire latency, the software overhead, and the message size divided by the bandwidth component.

The PLogP (for Parameterized LogP) model redefines message latency to be the user-observable message latency, or the time from which a send call is initiated on the source processor node to the time a receive call completes on the destination processor node. It also defines gap to be a function of message size. In this model, the parameterized gap includes the messaging software overhead that dominates effective short message bandwidth as well as sustained interconnection network bandwidth limits that apply to long messages.

One of the key parameter definitions that varies in these models is the definition of latency. Indeed, there are multiple aspects of latency that are important in a message passing system. To the applications programmer, end-to-end latency, or the time from when the

send call happens until the earliest time the first datum can be accessed on the receive node, is the latency the user sees. In this work, this end-to-end latency is commonly referred to as latency, and is consistent with the definition used in the PLogP model. Where appropriate, hardware or wire latency is used to refer to the latency of traversing through the interconnection network from one NI, through one or more switches, to another NI.

In this dissertation, the term overhead is broadly defined as the delay or lost compute ability (in terms of lost instruction issue slots) induced by cache misses, translation lookaside buffer (TLB) misses, copy operations, and other similar delays induced as a result of messaging. Here, messaging includes all operations related to message sends, message receives and message arrival notifications. In this model, some of this overhead is a part of the message latency since the message is actually placed on the network only after the overhead of setting up the message and copying the message into the network interface is incurred. In addition, the message data is not truly available to the application on the receive side until after the data is copied from the network interface into the user's address space, the destination user process is notified of the message arrival, and subsequently the first bytes of message data are loaded into the processor registers where the data can actually be used.

2.2 Understanding Message Overheads

Assuming a fixed network latency, end-to-end latency can be reduced by minimizing the time spent on both the send and receive nodes. It can also be hidden by allowing other useful work to proceed in parallel with message flight. Overhead can likewise be reduced by minimizing the amount of overhead required to send or receive a message, and allowing other useful work to proceed in parallel to message handling. This work focuses

on improving the nodes, or the end-points in the system architecture, and does not focus on the topology of the network itself or on the internal architecture of the network switches. There are many network topologies, each with advantages and disadvantages that make them more or less applicable in specific domains. Keeping the end-point architecture independent of the specifics of the network topology allows the nodes to be used as an efficient building block in many of these domains.

Since message latency is not all that different from memory latency, many of the techniques used to hide memory latency have been or can be used to hide message latency. Direct user-level access to the messaging hardware can be used to reduce both end-to-end latency as well as overhead. In addition to the traditional approaches of reducing message send and receive latency, reducing or eliminating kernel involvement in message arrival notification is beneficial in the reduction of end-to-end latency.

Local parallelism is useful for hiding some of the overheads in messaging systems. That parallelism is often provided in the form of an external communication or protocol processor. An example of such an external communications processor is the protocol processor in the FLASH [67] multiprocessor. While these extra processors can deal with much of the overhead, they do so at a cost. In addition to having the added complexity of a second, often special purpose processor, communication between the two local processors may introduce additional latency and overhead. To off-load a message send, the primary processor must first get the attention of the communications processor. The communications processor must also still communicate with the primary processor for message arrival notifications. As opposed to building a second special purpose processor and paying the energy and latency for the extra hop, it makes sense to just build the right processor.

Figure 1 shows an example of a message transmission on a machine that uses a kernel-mode network interface and traditional interrupts for message arrival notification. In these architectures, sends, receives, and notifications all make passes through operating system code. Since the operating system code competes with the user process for cache and TLB space, cache and TLB misses occur each time the system transitions from user to kernel space or from kernel space to user space. This results in additional overhead to the user process. On the left side of this figure is a representation of the activity on the sending node. Here the user code executes a send call. This send call ultimately results in a kernel call, so that the kernel can feed the message to the network interface. From there, the message is sent across the network through the network wires and switches. The right side of this figure represents the activity that occurs at the receiving node. The processor is executing user-level code when a message begins to arrive. The help of the kernel is requested by the NI via an interrupt. The kernel copies the message from the network interface to OS kernel memory. When the transfer is complete, the OS may notify the process that the message has arrived via a Unix signal or similar mechanism. Later, when the message is needed, the user application calls a receive routine that results in a system call to copy the message data from the kernel space into the user-space receive buffer.

User-level interfaces [12,41,44,50,86] and zero-copy protocols [20,33] significantly reduce the overhead of message sends and receives by eliminating operating system involvement and copying overhead. Figure 2 shows how a message send and receive may look on a machine that uses user-level network interface and traditional interrupts for message arrival notification. In such an architecture, sends and receives bypass the operating system. However, notifications still pass through the operating system code. On the left side

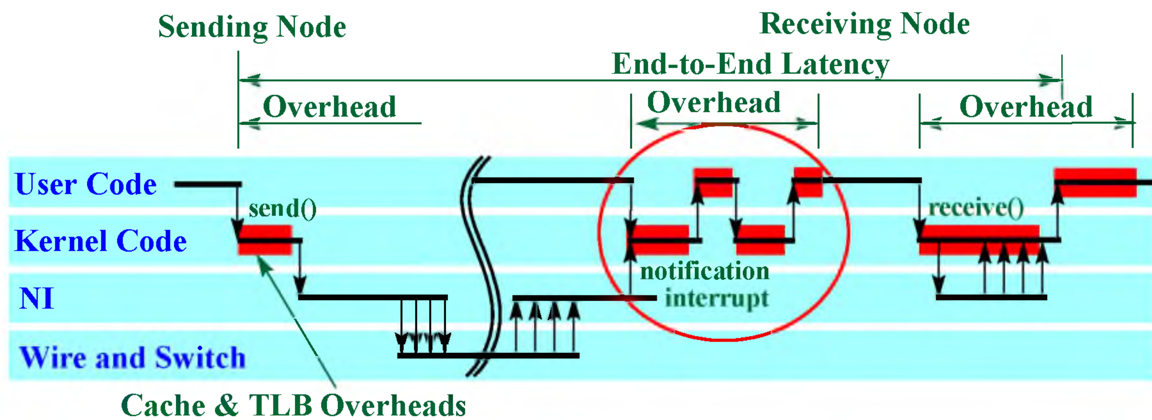

Figure 1: Anatomy of a message for a kernel-mode NI

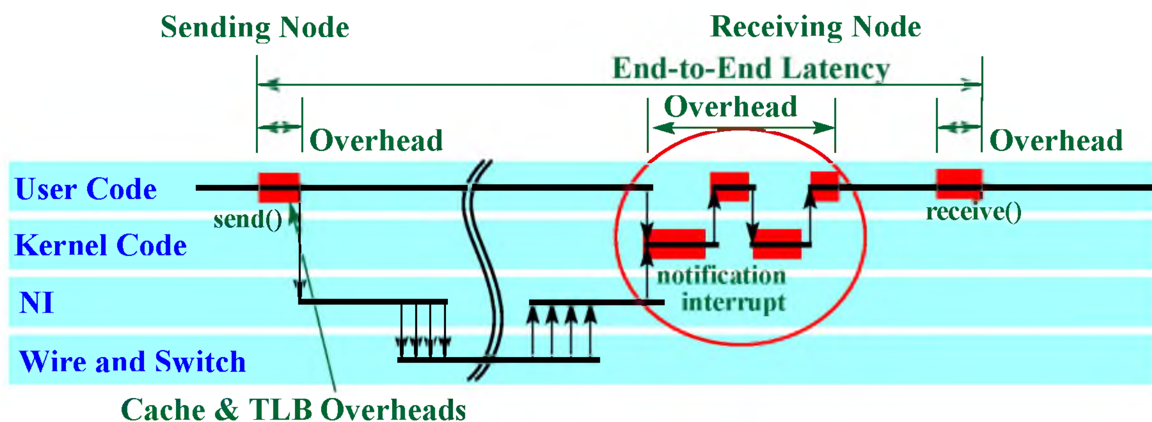

Figure 2: Anatomy of a message for a user-level NI without user-level notifications

of this figure is a representation of the activity on the sending node. The user code interacts directly with the network interface to deliver a message from the local user memory to the network. This may be accomplished via a library call; the kernel is not involved in the send. Similar to traditional architectures, the message is sent across the network through the network cables and switches. The right side of this figure represents the activity that occurs at the receiving node. The processor, as shown, is executing user-level code when a message arrives. As the message begins to arrive, the NI copies the message directly into the space of the destination user-process. Once the entire message has arrived, the NI notifies the kernel of the message arrival via an interrupt. The kernel then notifies the user code of the mes-

sage arrival via a Unix signal. When the message is subsequently needed, the user application can directly read the message from its local memory space, with no further kernel involvement.

With efficient message send and receive mechanisms in place, notifications become the primary performance and scalability bottleneck in both high-performance computing and general time-shared multiuser environments. Thus, streamlining notifications becomes the next significant optimization opportunity. Polling for notifications consumes significant processor and memory resources and is especially ill-suited for programs with irregular or unpredictable communication patterns. Interrupts in current architectures and operating systems are costly in terms of the number of processor cycles lost determining the source of the interrupt as well as handling and returning from the exception [45]. The high cost of interrupts makes them suboptimal for message notifications.

The overhead components of an interrupt-based notification include:

- processor pipeline flushing (due to the interrupt)
- serial instructions to get and save the processor state
- cache and TLB misses to bring in OS code and data to determine the cause of the interrupt
- reading NI registers or data structures to determine which process should be notified
- posting the notification to the process via a signal or other such mechanism
- cache, TLB and context switch overhead to begin execution of the user-level notification handler via the signal handler
- a trap to return from the user-level handler back to the OS
- serial instructions to restore processor state

- cache and TLB overhead to bring in OS code and data
- scheduler and context switch overhead to restore the original user process
- post kernel cache and TLB misses on the user process's instructions and data that were evicted by the kernel's memory references

As a part of this dissertation the overhead of servicing a network interrupt for a minimum sized packet was determined using a refined version of Schaelicke's interrupt measurement work [98]. Under Solaris 2.5.1 on a 147-MHz Ultra 1, such an interrupt takes approximate 119 microseconds (17500 cycles) when user-level code is utilizing the entire L2 cache. The process of handling such an interrupt results in approximately 380 kernel-induced L2-cache misses. (Fewer misses may be observed in practice if the user-level code is not utilizing the entire L2 cache). Assuming that each cache miss takes an average of approximately 270 ns to service [81], this accounts for about 103 microseconds or 87% of the interrupt processing time. The remaining 13% of the time is spent in flushing the pipeline after the interrupt and trap, carefully reading and saving critical processor state, querying the NI for information about the interrupt, and executing operating system code to determine how to respond to the interrupt. In addition to incurring the overhead of cache misses during an interrupt, the process that was running when the interrupt occurred could see up to another 380 L2 cache misses once it is rescheduled after the interrupt to refill the cache with its working set.

Other results indicate that servicing an interrupt may take anywhere from 20 to 100 microseconds on a contemporary machine [16,45]. In addition, the process of handling an interrupt causes the user process to incur 800-2,000 L1 and 450-1150 L2 cache misses for

a network induced interrupt [99]. While the clock-cycle times of modern machines have scaled, the number of misses has not reduced significantly.

The OS memory activity necessary to determine the cause of an interrupt and deliver the notification to the correct user-level process is by far the dominant overhead component that an interrupt style notification incurs. Since cache-miss related penalties scale at memory speeds, these penalties become even larger bottlenecks in terms of missed instruction issue as the memory gap widens. Optimizations to the OS and signalling system can reduce this overhead. However, reducing the number of cache misses and other overheads to get the OS penalty down below a few microseconds does not seem plausible in the near future. To make frequent notifications acceptable, the involvement of operating system must be significantly reduced or eliminated. The latency and overhead of the necessary remaining memory references in the notification process must be hidden by overlapping these memory references with other useful computation.

Schemes to combine polling and interrupts as a way to reduce these overheads, such as polling for a short time and then requesting an interrupt [68,77], have also been explored. They improve message-passing performance by improving the effective message-passing latency, or the amount of time that the user process perceives as message latency. However, these schemes do not directly improve the overhead associated with polling or interrupt handling. Instead, they attempt to balance those overheads to get some of the combined benefits of each. Rather than balancing these overheads, the right approach is to reduce and remove them by rethinking the notification mechanism. ULN does just that.

CHAPTER 3

ULN ARCHITECTURE

The key elements of the ULN architecture are an SMT processor, a message and thread synchronization table integrated into the core, a per-core on-die user-level NI, and a zero-copy messaging protocol. Each of these components and their benefit in the overall system architecture are described. This chapter also shows how ULN is optimized for efficient messaging, describes how notifications are delivered to the user-level process without kernel involvement, and walks through the path that a one-way message takes through ULN. Figure 3 shows the high-level block diagram of ULN and how the key components are connected.

3.1 The SMT Processor

This architecture employs a simultaneous multithreaded (SMT) processor, originally proposed by Dean Tullsen and Susan Eggers [111]. The SMT concept is seen in commercial processors such as the Intel Pentium 4, Atom, Itanium, and Core i7 processors, where it is known as HyperThreading [66], as well as in the IBM POWER5, POWER6, and POWER7 processors [57]. Such processors maintain several thread contexts that are all active concurrently. Multiple instructions are chosen by the processor core across the active contexts to execute in any given cycle. Instructions from independent threads may be issued on the same cycle. This allows the processor to execute two or more threads in parallel on

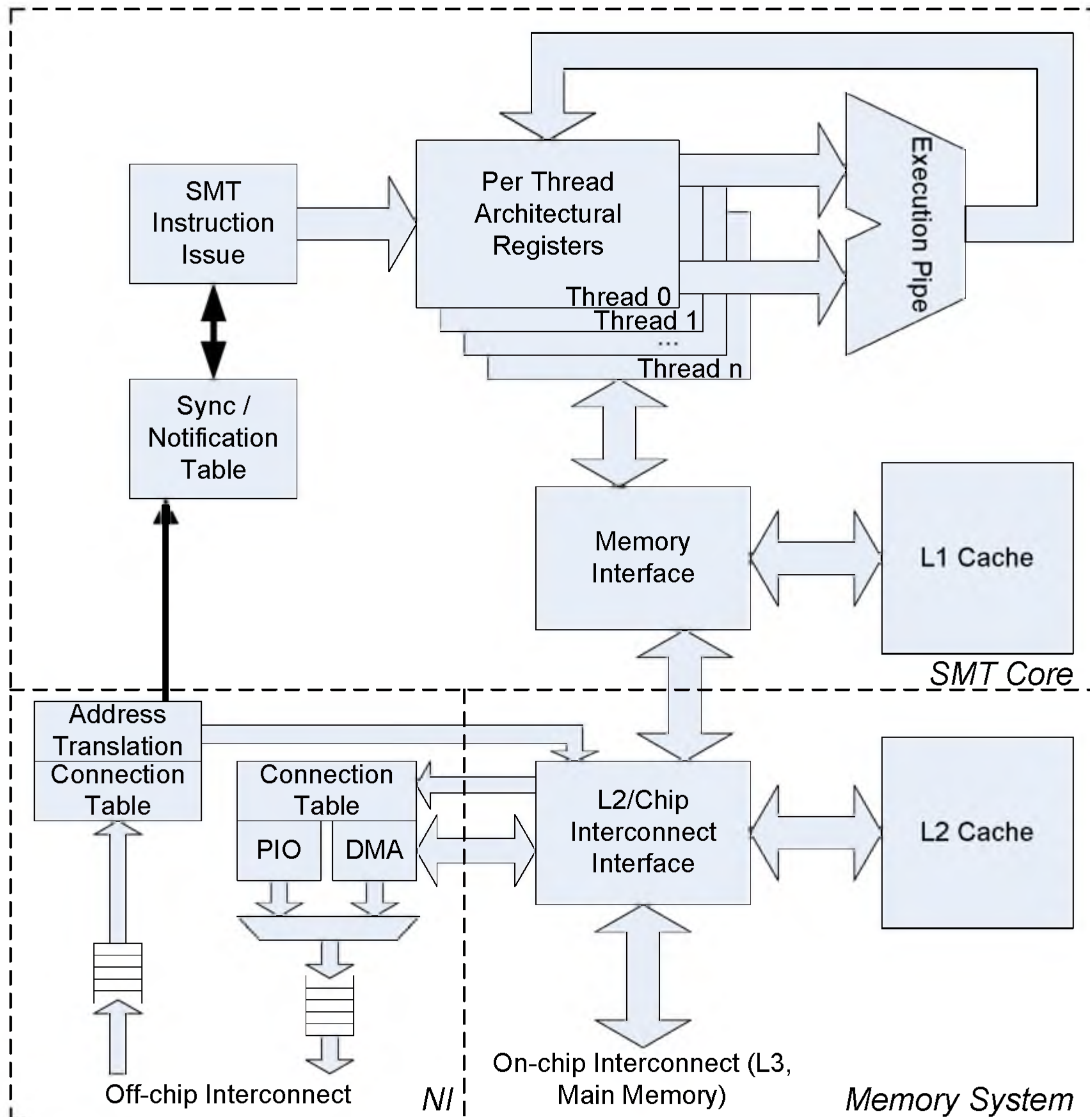

Figure 3: High-level ULN node architecture block diagram

a single core. It also allows for more efficient use of the available functional units, since if any single context cannot fully utilize the functional units, instructions from another thread context may fill the open slots.

By convention, in the ULN architecture one of threads on the SMT processor is commonly used as a dedicated communications thread. This thread generally handles message receives and notifications. This allows some of the remaining overheads associated

with messaging to overlap with computation. Specifically, using an SMT processor allows communication-related threads to run concurrently with computation-based threads. Since the communication thread generally has a different function unit usage signature from the compute thread, the SMT processor allows the compute threads to utilize compute function units while the communications thread interacts with the NI through the memory system.

The simultaneous nature of the SMT processor makes it possible to provide this overlap without the overhead and complexity of an extra communications processor. Furthermore, having both the communication and computation threads in the same core provides many opportunities for improved efficiency in communication between the computation and communication threads within the single node. This includes the effects of fast data communication via a shared first-level cache. It also allows for efficient thread synchronization through the mechanism described in Section 3.2.

3.2 User-level Notification and Synchronization Table

Part of the inefficiency of interrupt processing is due to the legacy view that interrupts are expected to be infrequent. In a fine-grained message-passing environment that uses interrupts for notifications, this is not the case. Having the NI tightly coupled to the CPU makes it possible for the NI to share some control over execution in much the same way as load-store units can control the pipeline as cache misses are detected.

Notifications are a way of telling the thread that it has work that is ready to be performed. In the case of message passing, it needs to deal with a message arrival. In a modern processor, when a cache miss occurs, the memory reference instruction stalls the processor. When the relevant data is returned from the memory system, the thread continues execution. Likewise, in the ULN message-passing architecture, mechanisms are provided to al-

low a thread to block, without the involvement of the operating system, waiting for a message to arrive. When a message arrives, the thread continues execution. This user-level notification mechanism is one of the main contributions of this work.

A synchronization table like that proposed by Dean Tullsen [110] allows for efficient communication between threads on the SMT processor. This synchronization table is extended to allow the NI to trigger events. In addition to threads being able to park and wake-up when triggered by a sister thread, they can park and wait for notifications from the NI on message arrival. A thread wishing to be notified when a message arrives can set a lock in a hardware synchronization lock table. This causes instruction issue to stall for that thread until it is unblocked by an arrival notification. In the common case, when the receive thread is scheduled, the NI just unblocks the relevant thread.

Arriving messages have several key pieces of information in their headers. These include a connection identifier that identifies the target receive process and a notification flag which specifies whether to release the associated synchronization table lock on arrival. If an existing handler thread is to be unblocked, the NI will attempt to unblock the thread by resetting the blocked bit in the table. This will fail if the relevant thread is not currently part of the CPU's active context. If so, the NI records the notification in a list and the OS wakes the relevant thread the next time the scheduler is run.

3.3 User-level Network Interface

Bringing the NI on die allows significantly more efficient interaction between the processor core and the NI [10]. Having the NI physically and logically closer to the processor, as opposed to the legacy model where the NI resides on an external IO bus, reduces the interaction latency and overhead. Allowing the NI to be in the coherence domain as op-

posed to on a noncoherent IO bus reduces a significant amount of software coherence overhead associated with sending and receiving data. This results in a reduction of message overhead. Furthermore, messages are sent out of and placed directly into the processor cache. This reduces effective message latency as well as the memory system overhead of fetching the message shortly after arrival.

3.3.1 Message Send Mechanisms

The ULN NI provides both programmed I/O (PIO) and direct memory access (DMA) mechanisms for sending messages. With PIO, message data is copied to the network interface explicitly by the sending program. With DMA, the sending process provides the NI with a message descriptor that gives the virtual address and length of the message. From there, the DMA engine in the NI fetches and sends the message in the background. PIO is appropriate for sending short messages, such as control messages, directly from registers without having to compose a message first in memory. DMA is more appropriate for larger messages or when messages are already composed in memory.

For both DMA and PIO transfers, message data is buffered in the NI and sent when there is enough data to make a full packet. This keeps the network from stalling for lack of message data. For PIO transfers, this can be due to any stall in the instruction issue stream, including a context switch. For DMA transfers, the memory system is more than capable of keeping up with the network demand. However, buffering is still necessary to cover any contention or stalls in the memory system.

Since messages are normally sent by the compute threads, if the CPU is busy copying message data to the network interface, it is less free to continue computing results. Placing DMA capability in the NI of a traditional design frees the CPU from the overhead of

shuttling data for larger transfers. Since sends are initiated directly from the user-level process, virtual addresses are supplied to the NI when requesting a DMA send. In addition, when data is received by the NI, it is placed in the user address space at the specified virtual address. The network interface includes the appropriate address translation capabilities, including a TLB and a page-table walk engine, to enable the use of these virtual addresses.

3.3.2 Message Arrival Mechanisms

When a message arrives, the NI looks up information about the connection in a connection table. Entries in the connection table contain a local context number as well as the virtual address base and bounds the message can legally target. In addition, the connection entry contains information on whether to inject messages into the cache. As a result of the connection table lookup, the NI has the necessary information to translate the virtual addresses in the message to physical addresses. It translates the address and combines the per-message injection hint with the per-connection message hint to determine whether to inject the message into the L2. If both the message and connection table agree to inject the message, the NI sends the message data to the L2 interface with a hint to inject the message into the cache. If the two disagree, the message is sent to the memory system with a hint not to inject the message into the cache.

Despite any hints, the NI does not inject message data into the L1 or L2 cache for any processes that are not currently running, to avoid unnecessary interference with the running process. If a message arrives without the injection bit set or if it arrives for a nonresident process but the corresponding addresses are already present in the cache, the data is just updated in that cache as opposed to evicting the line. Otherwise it is sent to lower levels

of the memory system to minimize interference with the currently running processes. Figure 4 illustrates the packet data injection process.

Generally connections will be marked in the connection table to allow a message to inject data into the cache. In this way, if a message proscribes cache injection, the message and connection table will agree and the message will be sent with the injection hint set. However, it may be desirable to have a large message that is not likely to be consumed immediately to bypass the cache so as to not evict live data. Additionally, the local operating

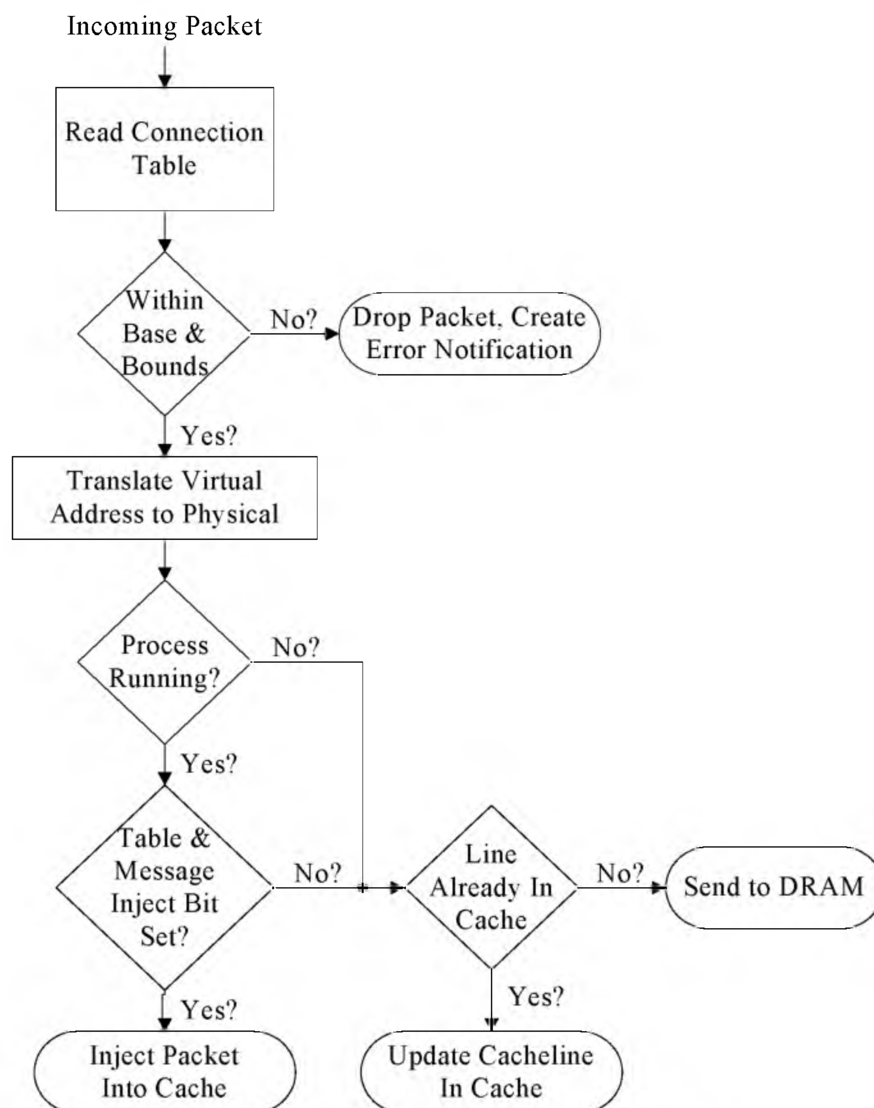

Figure 4: Packet data injection flow-chart

system may force messages not to be injected on a connection basis by clearing the injection bit in the connection table.

If directed to inject the message data into the cache, the L2 controller allocates a line in the cache and acquires exclusive ownership of the line. If the data injected covers a partial line, the cache fills the remainder of the line via a memory read. If a full line is written, the cache does not fetch the old contents of the line to avoid needlessly transferring data. This operation is the same as if the CPU were performing similar writes. If the L2 is not directed to inject the message, but the line is already in the cache, the line is not evicted, and is just updated. Otherwise, if the L2 controller is not directed to inject the message into the cache and the cacheline is not already present in the cache, the line is forwarded to lower levels of the memory hierarchy. If the cacheline injected is also present in the L1 cache, the L2 controller sends and update to the L1 cache to keep the contents coherent.

The primary advantages of this injection model are:

- In the cache injection case, it reduces overhead at the memory interface by saving the data two trips across the system memory bus, once to write the message and once to read it when used.
- It keeps the data near the CPU where it can be provided quickly on demand to reduce end-to-end latency.
- It avoids interference within the data cache when the current running process is different from the one the message targets.

3.4 Messaging Protocol

The message passing protocol used in this study is an extension of a sender-based protocol, similar to that used in Avalanche [33]. A sender-based protocol is a protocol

where the sender specifies where the message will be placed in the receiver's memory. Such a zero-copy message protocol allows messages to be delivered directly to user-space, thus avoiding the OS overhead of copying message data. In the Avalanche Direct Deposit Protocol (DDP) [107], as part of communications setup each node allocates a small fixed receive buffer that is pinned in physical memory. The send process is then permitted to write to any offset within the bounds of the buffer. In this way, the sender effectively owns and manages the remote receive buffer. This also serves to provide a simple end-to-end message flow control mechanism.

A sender-based protocol is used in order to reduce message latency and overhead, while simultaneously reducing overall hardware complexity. However, unlike with DDP, there are no restrictions on where a message can be placed in the receive node memory. Instead of specifying an offset within a receive buffer, the sending node specifies a receive side virtual address. When that message arrives at the destination node, the receive side NI checks the validity of the virtual address by first checking if the address lies within the bounds of the communication segment, performs a virtual to physical page translation, and places the data at the specified address anywhere within the user address space. This use of a sender-based protocol with no restrictions on the receive address provides the necessary mechanisms to avoid the overhead of copying data multiple times.

3.4.1 Message Descriptors

Message descriptors are the software-visible structure used for message send and receive information. The same message descriptor is used on both the source and destination side for simplicity. Some of the fields are only used on one side of the interface. On a message send, this structure is filled in by the user code and passed to the NI. On a message

receive, this structure is filled in by the receiving NI and placed in a message receive queue in user memory. The message descriptor is shown in Figure 5.

The destination identifier (dest_node_id) and source node identifier (src_node_id) in the message descriptor structure are job-specific virtual-processor identifiers. On the source side, the src_node_id is ignored by the NI and is normally not written. On the source side, the comm_id is supplied by the user-level code, but is checked for validity on the sender side NI. On the receive side interface, this is used to lookup the destination context as well as other protection information. The message_flags field contains the doorbell and cache_hint flags. The doorbell flag specifies whether the message should try to wake a thread upon arrival. The cache_hint flag allows the sender to specify whether to inject the message into the cache. The metadata_dest and data_dest fields are the virtual address where metadata and data should be placed on the receive side node. The metadata_size and dest_size fields describe the size, in number of bytes, of those portions of the message. The use of the metadata and data portions of the packet are determined by the programmer.

```
typedef struct
{
  uint32_t dest_node_id;        // Destination node identifier
  uint32_t comm_id;             // Identifies the communication channel
                                // Checked for validity at source
  uint32_t src_node_id;         // Filled in by the NI
  uint32_t message_flags;       // Notification method
                                // Where to inject into remote memory
  uint32_t user_field;          // User defined
  uint32_t metadata_size;       // Length of metadata
  uint64_t metadata_src;        // Local metadata pointer
  uint64_t metadata_dest;       // Remote metadata destination address
  uint64_t data_size;           // Length of data
  uint64_t data_src;            // Local pointer to data
  uint64_t data_dest;           // Remote data destination address
} MessageDescriptor_t;
```

Figure 5: Message descriptor structure

The message descriptor includes source-side metadata and data pointers, metadata_src and data_src, respectively. These fields are used on the message source side, along with metadata_size and dest_size, to describe to the DMA engine where the metadata and data that is to be sent exists in local memory. These fields are not communicated over the network. Instead, these fields are filled with zeros on the destination side. The metadata_dest and data_dest pointers are destination side virtual addresses. They are translated into physical addresses by the receiving NI. The user_field is just an arbitrary field that the programmer can use as desired.

3.4.2 Message and Packet Formats

On the wire, messages are composed of a message header, user-defined metadata, user-data and a per-message cyclic redundancy check (CRC). The message header consists of the relevant information from the message descriptor. The dest_node_id is a physical node number, translated by the outbound NIC. The comm_id in the network is trimmed to a physical width of 16 bits. There are 8 message flag bits, enough to encode the notification mechanism, the cache injection hints, and to leave room in the network protocol for expansion. The metadata_src and data_src are not transmitted over the network. The metadata_dest and the data_dest are both 64-bit pointers that are trimmed to 48-bits over the wire. Finally, the metadata_size and data_size are limited to 28 bits. This gives enough range to send a 512MB message. Larger transfers, if needed, could easily be composed of a series of these 512MB messages with minimal overhead.

The message format is shown in Figure 6. Upon message arrival, the NIC creates a message descriptor structure, shown in Figure 5. Relevant parts of the message descriptor are placed in the message header. The dest_node_id is replaced with the logical node at the

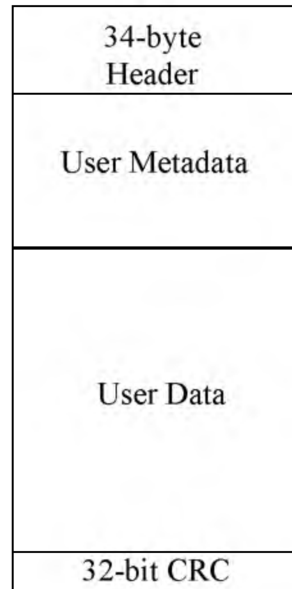

Figure 6: Message structure

destination end. The metadata_src and dest_src are set to zero on the receive side. The other fields, which are trimmed over the network, are zero-extended to fill the fields in the structure.

Messages are split into packets for efficient routing and transfer in the network. The packet header contains the physical destination node, the comm_id, a 16-bit packet sequence number, and a 32-bit packet CRC, for a total of 10 bytes of overhead. The maximum packet size chosen for this study is 1,034 bytes, 10 bytes of packet overhead plus a 1,024 byte payload. This results in a 1% physical protocol overhead on the network for a maximum size packet. Messages that are larger than 1,024 bytes in length are broken into multiple packets. The first packet in a message does not need to duplicate the destination node, or the comm_id. Messages that are shorter than the maximum packet length only occupy a packet of the appropriate size for the message on the network. Thus, a message with a zero bytes in the metadata and data fields occupies only 36 bytes on the network.

3.4.3 Connection Setup and Protection

Initiating communication in this architecture is accomplished via system calls that set up the virtual NI interface, coordinate the cooperating processes, and provide the communication identifier (comm_id in the structures above). Parallel job launch and communication setup are provided by a simple job launch utility. Protection between unrelated jobs is provided via the virtual memory system, and by checks and virtualization in both the source and destination network interfaces. User code uses virtual addresses for both local and remote side message data addresses. Node identifiers are virtualized and are checked and replaced with physical node numbers for routing purposes. The communication identifier (comm_id) is matched against context-specific user-inaccessible state in the NI.

The contribution of this dissertation is to define a general-purpose message-passing architecture for trusted peer processes, rather than to provide a secure, general-purpose internet-style protocol. A protocol such as UDP or TCP could be layered on top of this protocol, either directly or through a kernel interface, for facilitating communication between untrusted peers. This could be done by sandboxing the communicators, checking the communications in software, and providing software-based firewall mechanisms. These mechanisms, however, are not a part of this work.

3.5 Network Model

The network modeled in this dissertation is a simple point-to-point link with fixed latencies and bandwidths between nodes. Injection and ejection port contention are modeled. Contention within the internals of the network and details of the operation of the network internals are not modeled in detail. For reasonably well-behaved traffic patterns in

well-designed networks, congestion is not a significant limiter to injection bandwidth. In these cases, modeling input and output contention as well as a fixed network latency captures most of the relevant performance information for the network. Examples of large-scale system interconnection topologies with these properties include the folded Clos or multirooted high-radix fat-tree [25,72], flattened butterfly [62], and dragonfly [63]. In all of these, the network topology is relatively insensitive to a variety of reasonable traffic patterns. The message-passing architecture presented is deliberately architected not to be dependent on the network topology.

Network latency in a large-scale system with one of these high-radix is dominated by wire or optical fiber traversal times across the system. This is because the diameter of such networks is small, meaning that there are few router-to-router hops in the network. However, the cables connecting between the routers can be quite long. In a large system, cable length to reach from one end of the system to the other might be on the order of 40 meters. At two-thirds of the speed of light [47], signals take 200 ns to traverse 40 meters of optical fiber or electrical cable.

A fat-tree-topology made from a radix-64 router, such as the Cray YARC [101], can reach scales of 2,048 endpoints with only two levels of routers. Furthermore, scales of 65,536 endpoints can be connected with a three-level fat-tree. This implies that the maximum number of router chips that must be traversed to send a message is three for a 2,048 endpoint system and five for a 65,536 endpoint system.

Given that contention in the network is not critical, a flat fixed one-way interconnection network latency of 300 ns was assumed for the network in this evaluation. For the microbenchmarks presented in Chapter 4, the network latency can easily be factored out to

easily understand the latency and overhead at the endpoints. For the larger applications, this network latency ultimately limits the performance that an application may achieve.

3.6 The Journey of a Message

A typical message send in the ULN architecture consists of the following steps. First, the user-level program wishing to send a message composes the message metadata and data locally. A message descriptor is composed by filling in the dest_id, comm_id, message_flags, metadata_dest, metadata_size, data_dest, and data_size fields. In the case of a PIO transfer, the metadata_size and data_size fields are limited to an implementation-specific maximum size, not less than 1,024 bytes. In the case of a DMA transfer, the metadata_src and data_src fields must also be filled with a legal local virtual address.

In the PIO case, the user program writes the message header, metadata, and message data to the NI via memory-mapped PIO register writes. When all of the pertinent data has been transferred to the NI, the user program initiates a send via a load of the PIO register. This register returns a pointer to the status indicating whether the message was accepted and gives error information if it was not accepted. In the DMA case, the program uses an atomic swap to the NI's DMA register to provide a pointer to the message descriptor and to simultaneously get a status pointer.

The network interface begins the transfer by looking up the destination process and comm_id to check for permissions. If the destination is legal, the NI determines the route to the remote node, checks the pointers to the data to be transferred to ensure they are legal, begins to fetch the packet data, assembles the header of the message, and begins streaming the message to the network.

On the receive side, the message is placed directly in the address space of the receiving process by the NI. When the message header arrives, the NI checks base and bounds of the virtual address ranges in the message. As the data arrives, it is sent directly to the L2 interface with the appropriate cache hint. The L2 sends the message to lower levels of the memory hierarchy or acquires ownership of the relevant cache lines and injects the message into the cache as appropriate.

When the entire message has arrived, the NI fills in a notification structure in the user process's memory space, and if requested by the message, signals the SMT processor to unblock the relevant thread. If the associated thread is not currently in the CPU, the notification is still queued and a flag is set to let the OS know to unblock the thread the next time the relevant process is scheduled.

Figure 7 shows an example of a message in this architecture.

3.7 Summary

A user-level accessible NI is used to reduce send and receive overhead. Having the network interface physically and logically close to the processor opens up possibilities to more tightly integrate it with the processor core, further reducing overhead and latency. Having the NI on die gives the processor access to it on a per cycle basis. This close cou-

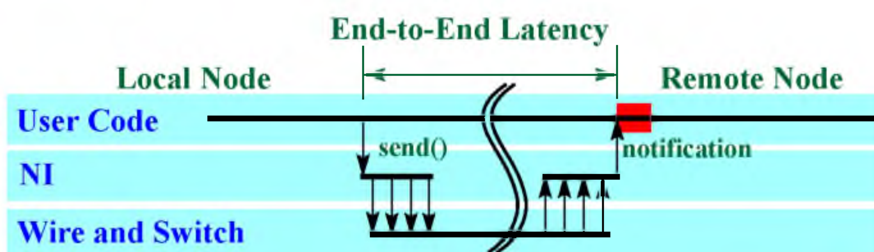

Figure 7: Anatomy of a message in the ULN architecture

pling further reduces the overhead in getting information to and from the NI. Message sends and receives do not have to go out over slow and inefficient I/O buses. A zero-copy protocol [20,33] is used to eliminate copying overhead for received messages. The combination of user-level access to a closely coupled NI and the zero-copy protocol allow for efficient sends and receives.

The ULN architecture presented significantly reduces send, receive, and notification overhead. The key features of the architecture are the following: SMT processors hide and tolerate message overhead and latency by allowing concurrency between communication operations in one thread and computation in other threads, send and receive overhead is reduced by a user-level network interface combined with efficient protocols, and notifications can be delivered directly to user-level without the overhead of an operating system. This architecture provides a low-latency, low-overhead message passing framework in which large-scale, fine-grained parallel applications can be efficiently executed across several processing nodes.

CHAPTER 4

MICROBENCHMARKS

A few microbenchmark kernels are used to show the key latency and overhead characteristics of this architecture for sends, polling receives, thread wake-up receives, and interthread communication. These characteristics are applied to a set of ping-pong kernels, used to illustrate the end-to-end latencies of messages in the ULN architecture. These ping-pong kernels are described in Section 4.3. The performance of multiple simultaneous threads in the SMT processor is analyzed in Section 4.4. These results are evaluated using the simulator described in Section 4.1. Section 4.5 applies these results to a symmetrical messaging kernel to understand the overheads of messaging in ULN. This kernel also serves as an example of how to use the message-passing features of ULN and illustrates a framework for a full communication system.

In addition to evaluating the kernels on the simulator, a few kernels are measured on real machines to characterize overheads. The Tuning and Analysis Utilities (TAU) [102] from the University of Oregon are used as described in Chapter 6 to characterize the Distributed Interactive Ray-Tracer (DIRT), a real-world computation and communication intensive application. These tools work by instrumenting the application source code, and as a result add some overhead. A simple kernel is used in Section 4.6 to characterize and account for that overhead. DIRT also has some internal instrumentation based on `gettim-`

`eofday()` that is used for understanding general performance characteristics. TAU and `gettimeofday()` overheads are measured in Section 4.6 and Section 4.7, respectively, on the same machine used to characterize the application. This machine is described at the beginning of Section 4.6.

This chapter describes each of those microkernel benchmarks in turn along with the associated results from those benchmarks. Those results are used in subsequent chapters to model and project the performance of ULN on a real-world application.

4.1 Simulator

A simulator was built to extract characteristics for use in the modeling of the larger full-system application model. The simulator is an execution-driven, cycle-accurate, SMT capable system simulator, known as SMT-MLRSIM. SMT-MLRSIM is based on ML-RSIM [98] Mike and Lambert's variant of the Rice Simulator, RSIM [94]. ML-RSIM adds a number of features to the baseline RSIM. ML-RSIM includes a first-level instruction cache (I-cache) model, translation lookaside buffer (TLB) models, a privileged mode model and I/O component models such as a PCI bridge model, a real-time clock model, and a hard disk drive model. The privileged mode extensions include the traditional privileged levels of execution as well as privileged instructions and privileged control registers to enable and disable interrupts, to manipulate TLB state, and to handle context switches. These extensions allow a full operating system to be accurately simulated.

SMT-MLRSIM adds SMT processor capability as well as the network interface described in Chapter 3 to the MLRSIM simulator. SMT-MLRSIM runs unmodified Solaris binaries under a modified NetBSD kernel, called lamix, when running in single-threaded mode. The kernel does not have the appropriate kernel locks to support a multithreaded pro-

cessor, hence lower-level programming is used when exploring multiple thread experiments.

SMT-MLRSIM is configured to simulate a system with the following characteristics:

- SPARC V8+ [113] instruction set
- 3 GHz core clock
- A MIPS R10000 [112] microarchitecture
- 2-4 SMT threads
- Up to 8 instructions issued per cycle
- 64 KB L1 instruction cache
- 64 KB L1 data cache
- 16 MB unified L2 cache
- 1 GB/s interconnection network injection bandwidth per core
- 300 ns interconnection network latency

Results from running kernels in this simulator are presented in the respective section below.

4.2 Messaging Overheads

Message sends, polling receives, notification-based receives, and interthread wake-up were examined on the simulator to understand the cost of each operation. These operations were measured by looking at the instruction trace from running the code in Section 4.3 below. Table 1 shows the timing for a message send based on a polling receive.

The send operation sets up a message descriptor, does an exchange operation to the NIC to initiate the send by giving the NIC a pointer to the message descriptor and getting

Table 1: Polling message timing

Event	Time
Send call initiation	T=0 ns
Send exchange issues	T=5 ns
Exchange arrives at NIC, NIC initiates send	T=15 ns
Exchange result returned to send processor	T=25 ns
Send call returns	T=26 ns
Message begins injecting at source	T=45 ns
First byte arrives at destination	T=345 ns
Last byte arrives at destination	T=383 ns
Last missing poll arrives at destination NIC (on average)	T=413 ns (+/- 10 ns)
Destiation NIC marks message as arrived	T=423 ns
Successful poll arrives at NIC (on average)	T=434 ns (+/- 10 ns)
NIC poll response returned to processor (on average)	T=444 ns (+/- 10 ns)
Receive call returns (on average)	T=448 ns (+/- 10 ns)

a status in return, and then checks the send status before returning. The send operation executes 31 instructions and involves an exchange with the NIC that takes 20 ns for the round-trip time. The the total send call time is 26 ns.

Leading up to the exchange instruction that initiates the send, the send routine executes 23 of those 31 instruction in about 5 ns. So the NIC receives the send command 15 ns into the send operation, including the 10 ns one-way trip time to the NIC. The NIC begins placing data on the wire 30 ns after the send command is received at the NIC, or 45 ns after the start of the send call.

The polling receive reads a NIC register to see if a message is ready, and then if a nonzero message descriptor pointer is returned, the poll is a success and that pointer is returned. If it is zero, indicating no message received, it reads the NIC register again. In the case that the message is there when the poll message is called, 34 instructions are executed taking 31 ns, including a 20 ns round trip to the NIC. In the case that the message is not

there, the maximum serialized polling frequency is 21 ns, set primarily by the 20 ns round-trip to the NIC. If a message arrives just in time to catch the cycle where the NIC is polled, the additional latency added to the message receive is the 10 ns response from the NIC, plus 4 ns to execute the code following the poll, or 14 ns. On average, however, the message will come in half-way through the polling cycle, thus the expected postmessage-arrival overhead is about 25 ns.

A message arrives from the NIC perspective 40 ns after the tail of the message is received on the wire. This is the effective pipeline latency of the NIC. For a minimal-sized message of 38 bytes on the wire, there is 38 ns of serialization latency between the head of the message arriving and the tail of the message arriving. Thus the minimal sized message is marked as arrived by the NIC 78 ns after the first byte arrives. For polling, the expected time before the processor notices the message is 93 ns after the first message byte reaches the NIC input. For a nonminimal-sized message, this also includes the serialization latency of the data and metadata portions of the message at 1.01 ns per byte over the 1 GB/s per core network, including packet overheads.

The notification-based receive, shown in Table 2, is similar to the polling-based receive with the exception that instead of repeatedly polling, it sets the wake mechanism, checks for an arrived message, and then sleeps on the mechanism. In the case that the message has already arrived, the receive call executes 36 instructions in 32 ns. In the case that a message has not arrived, there is a 28 ns setup overhead. When the message arrives, it takes the NIC 10 ns to notify the core and wake the receive thread. The thread then requires 25 ns to receive the message, for a total of 35 ns post message arrival, or 103 ns after the first byte of a minimal-sized message arrives at the input to the NIC. Thus the notification-

Table 2: Notification-based message timing

Event	Time
Send call initiation	T=0 ns
Send exchange issues	T=5 ns
Exchange arrives at NIC, NIC initiates send	T=15 ns
Exchange result returned to send processor	T=25 ns
Send call returns	T=26 ns
Message begins injecting at source	T=45 ns
First byte arrives at destination	T=345 ns
Last byte arrives at destination	T=383 ns
Destiation NIC marks message as arrived	T=423 ns
NIC wakes receive thread	T=433 ns
Receive thread completes receive of message	T=458 ns (+/- 10 ns)

based mechanism takes an additional 1 ns of overhead in the case that a message is already there and adds an additional 10 ns of latency in the case that a message has not already arrived. However, this small penalty comes at the benefit of reduced overhead from not polling in the case where the receive waits for some period for message arrival.

The thread wake-up overhead in the simulator at the lowest level is equivalent to a register write. However, for software convenience, this is wrapped by a function call. As the function call overhead is minimal, it takes 7 instructions and 2 ns to complete. The time until the thread wakes, however, and begins graduating instructions is 5 ns. This time is relatively low as the instructions are already fetched and decoded, hence instructions begin to issue immediately after wake-up. Table 3 summarized these overheads and latencies.

4.3 Ping-Pong

Ping-pong [52,53] message tests are commonly used in the characterization of a message-passing architecture. A ping-pong microbenchmark sends a minimal sized message round-trip from a source node to a destination node and back, which is useful for un-

Table 3: Message-passing latencies and overheads

Parameter	Value
Send Overhead	26 ns
Send Initiation Latency (send call to NIC data out)	45 ns
Polling Receive Overhead (message already waiting)	31 ns
Polling Receive Latency (no message waiting)	93 ns
Notification Receive Overhead (message waiting)	32 ns
Notification Receive Latency (no message waiting)	103 ns
Thread Wake-Up Overhead	2 ns
Thread Wake-Up Latency	5 ns

derstanding end-to-end message latency and message send, receive, and notification overhead. Consider two nodes, A and B. Node A sends a message to node B and then expects a response. Node B, waits for a message from node A and then immediately responds by immediately returning a message to node A.

Since messages in ULN can be received either by polling or by posting a user-level notification, both styles are demonstrated. The operation of the two styles is conceptually the same, with the only difference being the receive style. Additionally, the envisioned use of the multiple threads in the ULN architecture is to use one thread as a receive communications thread. A third style of ping-pong is to setup a receive thread on both nodes and have the receive thread on node B respond to the ping, and then have the receive thread on node A wake the initiating thread upon arrival of the pong message. In all three of these versions, node A and B first synchronize by doing a ping-pong before measurements are taken. This is to ensure that the message from node A is not sent before node B is waiting to receive it, so that round-trip latency measured does not include arbitrary synchronization overhead.

The common **`main()`** function for the two single-threaded versions is shown in Figure 8. This function starts with establishing the connection between two nodes in the **`establish_communication()`** call. It then calls the appropriate function for the node, **`nodeA_side()`** or **`nodeB_side()`**. After the tests in those functions are complete, it tears down the connection by calling **`teardown_communication()`**.

A helper function called **`sync()`**, shown in Figure 9, is also shared across these ping-pong tests. It is a simple routine that ensures the other node is ready before the measurements begin. Here, a polling message receive is used to keep the code similar to the receive style used in the first ping-ping test. The choice of the messaging style in this sync function does not impact the results as the purpose of the function is to ensure that the other node is ready before measurements are made on the real ping-pong test.

4.3.1 Polling-Based Ping-Pong

For the polling ping-pong test, the function **`nodeA_side()`** shown in Figure 10, is executed on node *A*. The function **`nodeB_side()`**, shown in Figure 11, is executed on node *B*. After initializing variables, both sides call **`sync()`** to perform an effective barrier

```
int main()
{
  CommID_t comm_id;

  // establish communication with the other node, exchange receive
buffer information, etc.
  comm_id = establish_communication(2);

  if(my_node_id() == nodeA)
    nodeA_side(comm_id);
  else
    nodeB_side(comm_id);

  teardown_communication(comm_id);
}
```

Figure 8: Ping-pong microbenchmark: `main()`

```
void sync(CommID_t comm_id, uint32_t remote_node)
{
  MessageDescriptor_t request;

  request.comm_id = comm_id;
  request.dest_id = remote_node;
  request.doorbell = NOTE_POLL;
  request.metadata_size = 0;
  request.metadata_src = (uint64_t) NULL;
  request.metadata_dest = (uint64_t) NULL;
  request.data_size = 0;
  request.data_src = (uint64_t) NULL;
  request.data_dest = (uint64_t) NULL;

  send(&request);
  poll_recv();
}
```

Figure 9: Ping-pong microbenchmark: `sync()`

operation with the other node to make sure the node is ready. In **`nodeA_side()`**, node *A* sends a ping message and immediately polls, or busy waits, for a return pong message to arrive. In **`nodeB_side()`**, node *B* polls waiting for the initial ping message from node *A* and when received, immediately returns a pong message. Round trip latency is measured on node *A* just before it sends the ping message to node *B* and ends just after it receives the pong message from node *B*. This gives us the minimal round-trip message time that would be expected for any communication. Though it takes advantage of the optimized sends and receives in the ULN architecture, it does not utilize the efficient notification mechanisms.

Again, referring to Table 1, on node *A*, the send call occurs at what we will call time T=0 ns. Data begins going out at time on the wire 45 ns after the call to **`send()`**. The network delivers the packet after a 300 ns latency and the first byte arrives at node *B*'s input at time T=345 ns. The message is received by node *B*'s NIC 78 ns after the first byte hits node B's input, or time T=423 ns. By varying when node *B* began polling, we can see the receive call return at anywhere from time T=438 ns to time T=458 ns, with an expected

```
void nodeA_side(CommID_t comm_id) // Node A code
{
  // Misc variables
  unsigned long long start, stop;
  // Message system defined types
  MessageDescriptor_t request;
  MessageDescriptor_t *response_ptr;
  // Example User defined type
  user_request_header_t user_request;

  // Specify communication
  request.comm_id = comm_id;
  request.dest_id = nodeB;
  request.doorbell = NOTE_POLL;
  user_request.message_type = ECHO_REQUEST;
  request.metadata_size = sizeof(user_request);
  request.metadata_src = (uint32_t) (caddr_t) &user_request;
  request.metadata_dest = KNOWN_NODE_B_LOCATION;
  request.data_size = 0;
  request.data_src = (uint64_t) NULL;
  request.data_dest = (uint64_t) NULL;

  // Sync up before measuring latency
  sync(comm_id, nodeB);
  // measure round trip latency
  start = get_clock();
  send(&request);              // send the ping message to nodeB
  response_ptr = poll_recv(); // wait for the pong message
  stop = get_clock();
  free_note(response_ptr);
  printf("Node A end-to-end latency is %d clocks\n", stop - start);
}
```

Figure 10: Poll-based ping-pong: `nodeA_side()`

one-way latency of 448 ns. The same thing then happens in the other direction. Thus the expected round-trip ping time is 896 ns +/- 20 ns.

4.3.2 Notification-Based Ping-Pong

Other than the difference in notification, the code and operation, the notification-based ping-pong is identical to the polling version. The **`nodeA_side()`** and **`nodeB_side()`** functions are nearly identical to the polling version, with the exception of using the **`NOTE_WAKE`** doorbell instead of the **`NOTE_POLL`** doorbell and the use of

```
void nodeB_side(CommID_t comm_id) // Node B
{
  // Message system defined types
  MessageDescriptor_t *request_ptr;
  MessageDescriptor_t response;
  // Example User defined type
  user_response_header_t user_response;

  // Specify communication
  response.comm_id = comm_id;
  response.dest_id = nodeA;
  response.doorbell = NOTE_POLL;
  user_response.message_type = ECHO_RESPONSE;
  response.metadata_size = sizeof(user_response);
  response.metadata_src = (uint32_t) (caddr_t) &user_response;
  response.metadata_dest = KNOWN_NODE_A_LOCATION;
  response.data_size = 0;
  response.data_src = (uint64_t) NULL;
  response.data_dest = (uint64_t) NULL;

  // Sync up
  sync(comm_id, nodeA);
  // Wait for a request from nodeA and respond
  request_ptr = poll_recv();
  send(&response);
  free_note(request_ptr);
}
```

Figure 11: Poll-based ping-pong: `nodeB_side()`

`sleep_recv()` instead of `poll_recv()`. These functions are shown in Figures 12 and 13. The differences are highlighted in bold. These changes cause the node to suspend waiting for the arrival of the ping and pong messages, respectively. When the message from node *A* arrives at node *B*, the thread wakes up, handles the receive, and returns a pong message. Likewise, after node *A* sends the initial ping message to node *B*, it sleeps until the pong message is returned. This measures the minimal additional overhead of handling a message receive with ULN's notification mechanism over polling.

Likewise, by examining the instruction trace, we can see that notification ping-pong has identical timing up until the notification. However, the notification time happens as a result of a thread wake-up and is independent of exactly how far in advance the thread goes

```
void nodeA_side(CommID_t comm_id) // Node A code
{
  // Misc variables
  unsigned long long start, stop;
  // Message system defined types
  MessageDescriptor_t request;
  MessageDescriptor_t *response_ptr;
  // Example User defined type
  user_request_header_t user_request;

  // Specify communication
  request.comm_id = comm_id;
  request.dest_id = nodeB;
  request.doorbell = NOTE_WAKE;
  user_request.message_type = ECHO_REQUEST;
  request.metadata_size = sizeof(user_request);
  request.metadata_src = (uint32_t) (caddr_t) &user_request;
  request.metadata_dest = KNOWN_NODE_B_LOCATION;
  request.data_size = 0;
  request.data_src = (uint64_t) NULL;
  request.data_dest = (uint64_t) NULL;

  // Sync up before measuring latency
  sync(comm_id, nodeB);

  // measure round trip latency
  start = get_clock();
  send(&request);              // send the ping message to nodeB
  response_ptr = sleep_recv(); // wait for the pong message
  stop = get_clock();
  free_note(response_ptr);
  printf("Node A end-to-end latency is %d clocks\n", stop - start);
}
```

Figure 12: ULN-based ping-pong: `nodeA_side()`

to sleep. Hence, the one-way message time is 458 ns and the round-trip ping-pong time is 916 ns.

4.3.3 Receive Thread Ping-Pong

In the ULN architecture a separate communications thread will commonly be used to handle incoming communications. A third variant of ping-pong explores message passing with multiple threads per node. Here, node *A* has two threads, a thread that mimics a compute thread, shown in Figure 14, and a simplified receive communications thread,

```
void nodeB_side(CommID_t comm_id) // Node B
{
  // Message system defined types
  MessageDescriptor_t *request_ptr;
  MessageDescriptor_t response;
  // Example User defined type
  user_response_header_t user_response;

  // Specify communication
  response.comm_id = comm_id;
  response.dest_id = nodeA;
  response.doorbell = NOTE_WAKE;
  user_response.message_type = ECHO_RESPONSE;
  response.metadata_size = sizeof(user_response);
  response.metadata_src = (uint32_t) (caddr_t) &user_response;
  response.metadata_dest = KNOWN_NODE_A_LOCATION;
  response.data_size = 0;
  response.data_src = (uint64_t) NULL;
  response.data_dest = (uint64_t) NULL;

  // Sync up
  sync(comm_id, nodeA);
  // Wait for a request from nodeA and respond
  request_ptr = sleep_recv();
  send(&response);
  free_note(request_ptr);
}
```

Figure 13: ULN-based ping-pong: `nodeB_side()`

shown in Figure 15. The compute thread on node *A* sends a ping message to node *B* and then suspends itself pending a wake-up event from the incoming communications thread running on the same node. Node *B* uses only a single thread, shown in Figure 16, that mimics a communication thread. When node *B* receives the ping message, the communications thread wakes up, handles the ping message and responds with a pong message. When the pong message arrives back at node *A*, the incoming communications thread wakes up to inspect the incoming message. It determines that the incoming message is destined for the compute thread and signals that thread to wake-up. Finally the compute thread on node *A* receives the message.

```
void nodeA_side(CommID_t comm_id) // Node A code
{
  // Misc variables
  unsigned long long start, stop;
  pthread_t response_thread;
  MessageDescriptor_t request; // Message system defined types
  user_request_header_t user_request; // User defined type

  // Spawn response handler thread
  assert(pthread_create(&response_thread, NULL,
         response_handler_task, (void *) comm_id) == 0);

  // Specify communication
  request.comm_id = comm_id;
  request.dest_id = nodeB;
  request.doorbell = NOTE_WAKE;
  user_request.message_type = ECHO_REQUEST;
  request.metadata_size = sizeof(user_request);
  request.metadata_src = (uint32_t) (addr_t) &user_request;
  request.metadata_dest = KNOWN_NODE_B_LOCATION;
  request.data_size = 0;
  request.data_src = (uint64_t) NULL;
  request.data_dest = (uint64_t) NULL;

  // Set this global variable before sync - helper will read it
  main_thread = get_thread_handle();

  // sync with the comm thread then the other node
  local_sync();
  sync(comm_id, nodeB);

  stage_sleep();  // initialize wake table

  // measure round trip latency
  start = get_clock();
  send(&request); // wake ping thread to do send
  thread_sleep(); // sleep until the response handler wakes us
  stop = get_clock();

  free_note(response_ptr);
  assert(pthread_join(response_thread, NULL) == 0);
  printf("Node A end-to-end latency is %d clocks\n", stop - start);
}
```

Figure 14: Multithreaded notification-based ping-pong: `nodeA_side()`

```
void *response_handler_task(void *arg)
{
  // Misc variables
  CommID_t comm_id = (CommID_t) arg;

  local_sync();

  response_ptr = sleep_recv();  // wait for the pong message

  // wake the main thread
  wake_thread(main_thread);

  pthread_exit(NULL);
}
```

Figure 15: Multithreaded notification-based ping-pong: communications thread

```
void nodeB_side(CommID_t comm_id) // Node B
{
  // Message system defined types
  MessageDescriptor_t *request_ptr;
  MessageDescriptor_t response;

  // Example User defined type
  user_response_header_t user_response;

  // Specify communication
  response.comm_id = comm_id;
  response.dest_id = nodeA;
  response.doorbell = NOTE_WAKE;
  user_response.message_type = ECHO_RESPONSE;
  response.metadata_size = sizeof(user_response);
  response.metadata_src = (uint32_t) (addr_t) &user_response;
  response.metadata_dest = KNOWN_NODE_A_LOCATION;
  response.data_size = 0;
  response.data_src = (uint64_t) NULL;
  response.data_dest = (uint64_t) NULL;

  // Sync up
  sync(comm_id, nodeA);

  request_ptr = sleep_recv();
  send(&response);

  free_note(request_ptr);
}
```

Figure 16: Multithreaded notification-based ping-pong: `nodeB_side()`

The multithreaded version of ping-pong illustrates one side of how two cooperating compute nodes may request services from each other. An example of such services may include requesting and returning data in a software shared memory system implemented on a cluster. It also illustrates the basic framework from which a full communications thread that handles incoming messages for several compute threads may be structured. It also allows the overhead of coordinating message events between a communications thread and a computation thread on a single node to be measured.

As the OS for the SMT-MLRSIM simulator does not support multiple threads, this code was run under a simpler development emulation environment and pieces were extrapolated to the simulator. In this case, the operation of the measured part of node *A* and all of node *B* are identical to that presented in Section 4.3.2. The primary difference is the addition of the interthread wake-up. The call to `stage_sleep()` sets a trigger for the interthread wake-up. This just clears the wake-up bit and should be done before the wake-up may be posted to avoid a race condition. We place this trigger outside the measured section, though the overhead is a small function call that executes in approximately 2 ns. The more critical operation is the interthread wake-up latency, which is about 5ns. Thus, this ping-pong variant requires an additional 5 ns over the single-threaded notification-based ping-pong for a total of 921 ns round-trip.

4.4 SMT Performance

An understanding of how fast multiple threads execute simultaneously on the SMT processor is needed to make a complete model of real-world applications. The simulator was used to measure the per-thread speedup with multiple threads running on a hand-written assembly kernel. The kernel performs a mix of operations including loads and stores as

well as computation. As the kernel is cooperative, there is little cache and memory interference as the number of threads increases, this model is optimisitic. The results of those runs are shown in Table 4.

The relative aggregate IPC can be thought of as a function that saturates as the architecture schedules the maximum number of instructions afforded by the workload's instruction mix. This model can be used to model the SMT aggregate speedup by solving for the least-squares fit of the above data for a and b in the equation,

$$a\left(1 - e^{-\frac{n_T}{b}}\right)$$

Doing so yields,

$$3.20\left(1 - e^{-\frac{n_T}{2.67}}\right)$$

The data is graphed against the best-fit equation in Figure 17.

Table 4: SMT thread speedups

Number of Active Threads	Measured Aggregate IPC	Relative Aggregate IPC	Per-Thread Relative Speedup
1	0.778	1	1
2	1.36	1.75	0.876
3	1.67	2.15	0.717
4	2.00	2.58	0.644
5	1.95	2.51	0.502
6	2.35	3.03	0.504
7	2.24	2.89	0.412
8	2.39	3.07	0.384

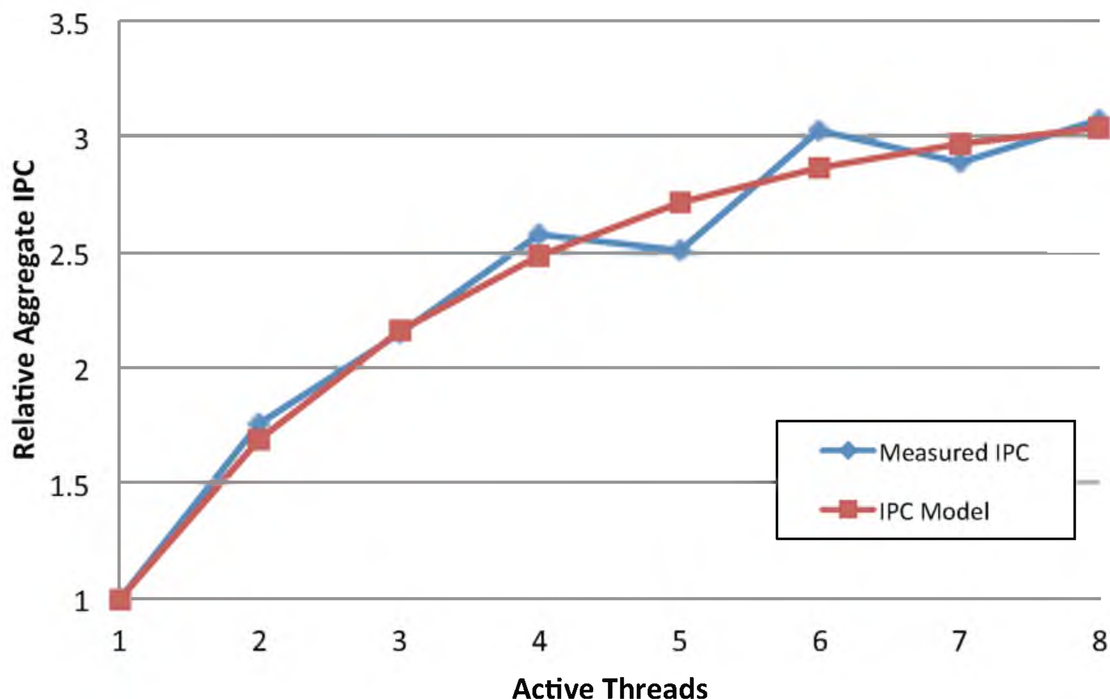

Figure 17: Measured SMT speedup versus best-fit model

As can be seen, the fit fairly closely matches the measured data. The model provides a smooth curve that does not include artifacts of the simulated microarchitecture. This thread speedup model is used for the application model in Chapter 7.

4.5 Polling versus User-Level Notification Overheads

A two-sided or symmetrical messaging kernel is used to show the envisioned use of the ULN architecture and to measure the overhead of polling. This microbenchmark builds upon the threaded version of ping-pong. Each node has two or more threads, one or more compute threads and a single communications thread. The communications thread responds to requests from the other nodes in the system, returning responses to messages and directs responses initiated by compute threads running on the local node to the requesting compute thread. The compute threads each execute local code for some amount of time and

periodically send a message to another node in the system and suspend, awaiting a response.

This version of the microbenchmark simulates and illustrates a simple but complete communications system in this architecture. In particular, it shows the framework from which a software shared-memory library for a cluster built around this architecture may be constructed. Alternately, this benchmark could be used as an example framework from which a simple service request or synchronization/coordination library could be constructed in this architecture.

By comparing the per-thread speedup of n threads with n+1 in threads, we can get an idea for how much of a performance penalty a polling communication thread can have on a number of compute threads. In the single compute thread case, the time penalty for polling is up to 18.5% for both the compute and communication threads. For 7 compute threads, the penalty is 11.6% compute and communication threads. The result is that the round-trip time for a minimal sized message with polling and thread notification increases from 881 ns to 886 ns, only 15 ns or 1.66% faster than the low-overhead 901 ns case of user-level notifications. For a model where the compute thread executes for some time, sends out a request and then waits for a response, the computation per message would have to comprise less than 81 ns of single thread work between the communications requests before polling even begins to make sense from a performance perspective. However, in this case, the energy for polling unproductively for the entire cycle instead of having the processor be actively dealing with message receives 3.16% to 3.50% of the time far outweighs the up to 1.66% performance advantage. This shows that notification-based receives should be used at all times in the ULN architecture.

4.6 TAU Measurement Overhead

To understand how much time is split between local computation, communication, and load-imbalance in a run of the DIRT application, the application was instrumented using TAU. The profiling information produced with the TAU tools is used to determine the time spent in various tasks.

Fully instrumenting the code with TAU adds nontrivial measurement overhead to the run data. To properly account for that overhead, measurements were taken on the TAU code itself. A small benchmark was constructed that measure the TAU overhead. This code is shown in Figure 18. A triple nested set of **TAU_PROFILE** calls were used in a 1,000,000-iteration outer loop to include many samples of the measurement overhead. A triple nest was used to ensure that the overhead was consistent across two calls and that nesting did not have an unexpected impact on the results. The benchmark code was run 100 times to ensure enough outer loop samples to determine that the tiny differences between the “loop level 1” and “loop level 2” measurements were not statistically significant.

```
#include <TAU.h>
#define ITER_COUNT 1000000
int main()
{
  {
    TAU_PROFILE("outer loop measurement", " ", TAU_USER);
    for(int i = 0; i < ITER_COUNT; i++)
    {
      TAU_PROFILE("loop level 1 measurement", " ", TAU_USER);
      {
        TAU_PROFILE("loop level 2 measurement", " ", TAU_USER);
        {
          TAU_PROFILE("loop level 3 measurement", " ", TAU_USER);
        }
      }
    }
  }
}
```

Figure 18: TAU instrumentation overhead measurement technique

The results measured on a real machine were gathered on a 120 node Dell PowerEdge 2650 based cluster. The nodes are connected via switched gigabit ethernet. Each node has two 2.4 GHz Intel Xeon 80532 processors. This is a Pentium 4 class processor employing a Prestonia core. Each processor has an 8KB L1 cache, a 512 KB L2 cache, and a 400 MHz front-side bus. The node has 2 GB of PC1600 DDR1 registered dynamic random access memory (DRAM) with a total of 3.2 GB/s system memory bandwidth attached via a ServerWorks Grand Champion LE chipset.

The average time of the “loop level 1” measurement and “loop level 2” measurements are 0.997 μs and 1.01 μs, respectively. The difference between the two is less than 1% and is well within the expected values of the measurements. The standard deviation of the two measurements are 0.0115 and 0.0126, respectively, indicating that there is not statistical difference between these two samples. The average between the two, or 1.00 μs, is used as the TAU measurement overhead in the subsequent performance calculations below.

The 1.00 μs overhead is a measure of the total overhead associated with each measurement. That overhead can be split into two parts: The exterior portion is the overhead associated with the TAU timer function before the time-stamp counter is sampled when starting a counter plus the overhead of the TAU timer function after the time-stamp counter is sampled when stopping a timer. The interior portion of the overhead is associated with the measurement between the two time-stamp counter samples. The inner loop, has an average measured time of 0.487 μs. This represents the interior overhead. The exterior overhead is then the difference between the total measurement overhead and the interior overhead. This is approximately 0.520 μs.

TAU reports the time spent in a function in terms of *inclusive* and *exclusive* time. The exclusive time of a function is the time spent in the function proper, excluding any time spent in children function calls. The inclusive time of a function is the total time from the invocation until the time the function call returns, including the time spent in any children function calls. To account for TAU overheads in measured codes, the interior and exterior overhead are removed from the function's measured time. For each function invocation, the interior overhead is subtracted from the function's exclusive and inclusive time. For each subroutine invoked, the TAU exterior time is subtracted from the caller function's exclusive and inclusive time. The number of invocations of each function and the number of subroutines called for each function is reported by TAU.

4.7 Manual Instrumentation Overhead

DIRT was also manually instrumented by its authors at a coarse-grain in select places using an internal set of timing and event counting macros. All of these macros are trivial, built using POSIX **`mutex()`** and where appropriate, **`gettimeofday()`**. The overhead of these macros were measured and subtracted from the TAU measured results. A simple kernel that repeatedly calls each of these macros was used to measure that overhead. The counting macros are shown in Figure 19 and the timing macros are shown in Figure 20.

The measurements were taken on the same machine described in Section 4.6. The overheads of **`countit()`**, **`countem()`**, **`timestamps()`**, **`timestampe()`**,

```
#define countit(a) {pthread_mutex_lock(&iolock);a++; \
                    pthread_mutex_unlock(&iolock);}

#define countem(a,d) {pthread_mutex_lock(&io-
lock);a+=d;pthread_mutex_unlock(&iolock);}
```

Figure 19: Manual instrumentation counter macros

```
#define timestamps(t0) \
struct timeval t0; \
if (gettimeofday(&t0, 0) != 0) \
  fprintf(stderr, "Error getting time\n");

#define timestampe(t0, te) \
{ struct timeval t1; \
  if (gettimeofday(&t1, 0) != 0) \
    fprintf(stderr, "Error getting time\n"); \
  pthread_mutex_lock(&iolock); \
  te += (t1.tv_sec - t0.tv_sec) + (t1.tv_usec - t0.tv_usec)*1.e-6; \
  pthread_mutex_unlock(&iolock); }

#define timestamps2(te) \
{ struct timeval t0; \
  if (gettimeofday(&t0, 0) != 0) \
    fprintf(stderr, "Error getting time\n"); \
  pthread_mutex_lock(&iolock); \
  te = t0.tv_sec+t0.tv_usec*1.e-6; \
  pthread_mutex_unlock(&iolock); }

#define timestampe2(t0) \
{ struct timeval t1; \
  if (gettimeofday(&t1, 0) != 0) \
    fprintf(stderr, "Error getting time\n"); \
  pthread_mutex_lock(&iolock); \
  t0 = (t1.tv_sec + t1.tv_usec*1.e-6 - t0); \
  pthread_mutex_unlock(&iolock); }
```

Figure 20: Manual instrumentation timing macros

`timestamps2()`, and `timestampe2()` are measured at 0.150 μs, 0.150 μs, 0.384 μs, 0.544 μs, 0.544 μs, and 0.544 μs per call, respectively.

4.8 Speedup of Modern Core

The analysis presented in Chapter 5 was performed on the same Pentium 4 machine described above. To understand how DIRT would perform on a more recent CPU core, a small workload appropriate for a single worker node was run on a single worker node of the Pentium 4 machine and a 3.4 GHz Intel Core i7-2600K Sandy Bridge-based machine. A single worker thread on the Pentium 4 machine took 1956 seconds to render 100 frames of this small scene, where a single thread on the Core i7 machine took 312 seconds. This is

an absolute speedup of 6.27. However, correcting this for the clock frequency difference gives an architectural or clock-per-clock speedup of 4.43.

Since the SMT processor in this work operates at a frequency of 3 GHz, the speedup is corrected by the ratio of 2.4 GHz to 3 GHz to give a speedup of 5.53.

4.9 Microbenchmark Summary

In summary, this section shows that the ULN architecture provides message arrival notification mechanisms that approximate the effective latency of continuous polling with little overhead. An example framework for a communication infrastructure is illustrated. The SMT processor in the architecture is characterized showing the speedup of multiple threads running simultaneously. The overheads of TAU and the manual instrumentation are characterized. These key speedups and overheads are used in the evaluation and forward projection of DIRT. These key parameters are summarized in Table 5.

These measurements and microbenchmark results are used in Chapter 6 to account for overhead in the analysis of DIRT and to produce the results presented in Chapter 8 on the application and architecture model presented in Chapter 7.

Table 5: Summary of key speedups and overheads

Parameter	Value
Send Overhead	26 ns
Send Initiation Latency (send call to NIC data out)	45 ns
Polling Receive Overhead (message already waiting)	31 ns
Polling Receive Latency (no message waiting)	93 ns
Notification Receive Overhead (message waiting)	32 ns
Notification Receive Latency (no message waiting)	103 ns
Thread Wake-Up Overhead	2 ns
Thread Wake-Up Latency	5 ns
Per-thread Speed-up 2 Threads	0.876
Per-thread Speed-up 3 Threads	0.717
Per-thread Speed-up 4 Threads	0.644
Per-thread Speed-up 5 Threads	0.502
Per-thread Speed-up 6 Threads	0.504
Per-thread Speed-up 7 Threads	0.412
Per-thread Speed-up 8 Threads	0.384
SMT Speed-up Model	$3.20\left(1-e^{-\frac{n_T}{2.67}}\right)$
TAU Interior Overhead	487 ns
TAU Exterior Overhead	520 ns
`countit()`/`countem()` Overheads	150 ns
`timestamps()` Overhead	384 ns
`timestampe()`/`timestamps2()`/`timestampe2()` Overheads	544 ns
Core i7 vs. Pentium 4 Speedup, corrected to 3 GHz	5.53

CHAPTER 5

REAL-WORLD APPLICATION: DIRT

The microbenchmarks presented in Chapter 4 show the characteristics of the ULN architecture. A real-world application is needed to show the benefits of the ULN architecture. The Distributed Interactive Ray Tracer, or DIRT [35,36], is a scalable interactive ray tracer used to visualize large volumes in a distributed computing environment. DIRT, produced by Demarle, is a cluster-based derivative of Steve Parker's Real-Time Ray Tracer (RTRT) [91,92,93]. This ray tracer is an example of a compute and communication intensive, real-world, scalable application that serves as a test case for the ULN architecture. This chapter describes the structure and operation of DIRT.

5.1 The Real-Time Ray-Tracer

RTRT is a highly scalable parallel interactive ray-tracing system designed for large cache-coherent nonuniform memory access (CC-NUMA) [73] distributed shared-memory (DSM) [1] multiprocessor systems, such as the SGI Origin 2000 [69] supercomputer. The system was designed to prove that a software ray-tracer could be used to interactively render and explore both conventional scene models as well as large complex data sets. It has been shown to scale to many hundreds of processors and has been used to render both analytical object models as well as multihundred gigabyte polygonal data sets, structured mesh volumes and unstructured mesh volumes. It can be used to visualize simple surface

models as well as to extract and visualize isosurfaces or maximum intensity projections from large data sets.

To be interactive, the ray-tracer must be capable of rendering many frames per second over these large data sets. Large shared-memory systems have sufficient aggregate memory to hold the entire data set in memory, enough computational resources to render the scene in real-time, and the necessary interconnection bandwidth to move the large data set to the appropriate computational resource. The shared memory also contains a shared hierarchical acceleration structure used to reduce the quantity of scene data that must be traversed to render, a shared work queue used to manage work assignments, and shared framebuffer used to store the rendered frames.

RTRT's architecture consists of a `Display` thread and a number of `Renderer` threads. Each one of these threads executes on its own CPU core. The `Display` thread manages the set of tasks required to render and display frames. For each frame rendered, it determines viewing information such as the camera position and direction, as well as the color and position of lights in the scene. It makes that information available to all of the `Renderer` threads by placing it in shared memory. The `Display` thread also generates a set of work assignments necessary to render the frame and places those assignments in a shared global work queue. At the start of each frame, the `Display` thread populates the work queue with all of the assignments necessary to render the current frame.

Each `Renderer` thread pulls a work assignment from the shared work queue. An assignment consists of a rectangular group of rays that pass from the camera model through the viewing window, known as primary rays. The `Renderer` thread processes that assignment, contributing the resulting image tile to the shared framebuffer. It processes each ray

by tracing the ray along its trajectory, through the large data set contained in the machine's shared memory. It writes the resulting piece of the frame's image, called an image tile, to the shared frame buffer. After rendering each tile, it returns to the work queue for another work allocation. If the queue is empty, it joins a completion-of-frame barrier. Each `Renderer` thread repeats this process until explicitly signalled to exit by the `Display` thread. When all of the image tiles for a particular frame have been written back to the frame buffer, the `Display` thread displays the resulting frame on the screen and repeats the process of determining the new viewing information and refills the work queue.

It is worth noting that the number of assignments exceeds the number of `Renderer` threads such that each thread will complete several assignments per frame. The work queue is generated with varying size work allocations. In particular, for each frame, it contains a number of large assignments in the queue followed by successively smaller assignments per frame. This results in the `Renderer` threads receiving larger assignments toward the beginning of the frame and successively smaller allocations toward the end of each frame. Compared to a fixed size allocation, this work distribution scheme minimizes the overhead of going to the queue an excessive number of times, yet results in small assignments toward the end of the frame resulting in good load balance.

5.2 The DIRT Derivative

Large CC-NUMA machines provide the mechanism needed to support fine-grained access to the acceleration volume hierarchy, the underlying scene data, the work queue, and the frame buffer. Unfortunately, scaling CC-NUMA architectures beyond a few thousand nodes becomes prohibitive in the amount of hardware state and coherence traffic required

to keep the caches coherent among all of the processors. To address this issue DIRT was created as a distributed memory variant of RTRT for use on clusters.

DIRT shares much of the same basic structure as RTRT. DIRT runs on a number of nodes. One of those is called the supervisor node. It runs the display thread, manages and distributes the scene state, manages and distributes the work allocations, and collects the rendered tiles. The remainder of the nodes are called worker nodes. They each run one or more `Renderer` threads.

However, there are differences. Since clusters do not implement a shared memory in hardware, DIRT implements a software shared memory across the worker nodes to store and manage the shared scene data. The work assignments are explicitly sent as messages from the supervisor node to the worker nodes, as opposed to being in a shared memory-based queue. The hierarchical acceleration structure is much smaller than the scene data and is replicated on each node. Finally, the framebuffer only exists on the supervisor node. All of the worker nodes explicitly communicate rendered tiles back to the supervisor node. As opposed to joining a barrier at the end of a frame, the worker nodes are unaware of frame boundaries. They repeatedly process assignments and return tiles. The supervisor knows that a frame is complete when all of the tiles are returned.

The scene data is split into blocks known as bricks. Each node in the system is considered the owner of some portion of the bricks and keeps those bricks in local memory. The node that owns a brick is referred to as the home node for that brick. A part of the memory on each node is also used to cache remote scene bricks. When a node in the system needs to access a brick, it does a check. If it is the owner of the brick, a pointer to the brick is satisfied out of local memory. If it is not the owner of the brick, the local brick cache is

checked to see if the local node already has a copy of the brick. On a miss in the brick cache, the local node sends a request message to the home node of the brick. The home node updates the list of nodes sharing the data and responds with the requested scene data. These request messages are said to be asynchronous in nature in that the home node does not know if and when data may be requested.

To facilitate the message passing for the control system as well as for the software shared memory system, each node also runs a `communicator` thread. This thread handles all incoming messages

5.3 DIRT Details

DIRT is a broad ray-tracing framework capable of rendering a variety of scenes, from traditional polygonal scene models to mathematical surface models, isosurface extraction or maximum intensity projections in structured and unstructured volume data. Isosurface extraction or visualization is used as the focus application of this work as is requires a mixture of local computation to compute intersections of the rays with an isosurface in the data as well as remote communication to share the large dataset among the nodes. This section focuses on the pieces of DIRT that are utilized to render isosurfaces present in such a dataset. Some real-world examples of such a visualization include examining structures in medical data or visualizing isovoltage surfaces in a large electric field. Chapter 6 describes the dataset and analyzes DIRT in the context of visualizing isosurfaces in a large generated 3-D volume.

5.3.1 Node Initialization

The supervisor node parses the command line, sets up the display output, checks for the existence of relevant scene files, creates the scene, and starts a local `communicator` thread. It then starts the jobs on the worker nodes. Next it waits at a barrier for all of the worker nodes to finish initialization and then sends out the initial scene state and work assignments to each worker node. On each worker node there is a `main`, a `communicator` thread, and some number of `Renderer` threads. The `main` thread is responsible for initializing the node and spawning the other threads. It initializes the software shared memory layer and reads in the local node's portion of the scene dataset. It then starts the `Renderer` threads and the `communicator` thread, builds the macrocell hierarchy and sleeps until the job is complete. The `communicator` thread handles received communication on the node. The application initialization time both on the supervisor and on the worker nodes is not considered important by the authors of DIRT and is ignored in this work.

5.3.2 Worker Node Operation

After initialization, the `Renderer` threads sit in a loop waiting for work assignments from the supervisor node. Since the `Renderer` threads are spawned before the initialization is complete, they sit and wait for a task assignment until after all nodes have performed initialization and have passed a barrier. This overhead is subtracted out of the `Renderer` thread runtime in Chapter 6.

Each assignment contains a variable number of pixels to render from a rectangular region of the screen. For each of the pixels, the `Renderer` thread traverses the local acceleration structure, known as the macrocell hierarchy, along the direction of the ray. The macrocell hierarchy is a 3-D volume hierarchy that, at each level, tracks a summary such

as the 3-D bounds as well as minimum and maximum values of the underlying data. The top-level of the hierarchy is a coarse structure, such as the 3x3x3 grid used in the scene in Chapter 6. Each of those top-level cells has a similar sized grid of macrocells under it. Macrocell hierarchies tend to be about 2 to 3 levels deep. The bottom level of the hierarchy maps to a number of scene bricks, which encapsulate the scene data.

Initially, the ray is checked against the top-level of the macrocell hierarchy. For the macrocells the ray passes through, if any, the target isovalue is checked against the macrocell's minimum and maximum values. If the value is between the minimum and maximum, the ray could potentially intersect with the target isosurface within that macrocell. In this case, the next level of macrocells in the hierarchy is checked. If the isovalue is outside the range of a macrocell, the macrocell and all underlying data can be skipped over, saving computation and data references.

If the isovalue is in the bounds of a particular macrocell at the lowest level of the hierarchy, the brick data is traversed along the ray. Since the macrocell hierarchy and brick data are traversed in order from closest to farthest along the path of the ray, once the ray intersects with the target isosurface, no further searching is required. In this case, the pixel corresponding to the ray is colored according to the scene parameters. If the ray misses the volume or does not intersect with the isosurface in the volume, the pixel is colored according to the background color in the scene.

5.3.3 Software Shared Memory System

As the brick data is traversed, the references to the brick data go through the distributed software shared memory. When a `Renderer` thread references shared memory, it firsts checks if the local node is the home for the data referenced. If it is, the reference is

satisfied directly out of local memory. If the local node is not the home node, the local software shared memory cache is checked to see if a local copy of the data is already cached. If the reference is in the local shared memory cache, it is satisfied from local memory.

If the reference is remote and misses in the local software shared memory cache, the `Renderer` thread sends a shared memory read request to the associated home node.The thread then blocks until the shared memory request is satisfied. The communication associated with these cache misses is the most interesting part of DIRT in the context of this work. The measured latency waiting for these cache misses to be filled by a remote node is significant. Furthermore, the overhead associated with processing these requests on both the local and remote node directly interferes with the performance of other `Renderer` threads on both nodes.

The `communicator` thread running on each node handles all incoming message traffic. For incoming scene data, it updates the local copy of the scene state. For work allocations, it places assignments in a node-local work queue. The `Renderer` threads receive assignments from this work queue. For shared memory messages, the `communicator` thread executes the DataServer class that implements the software shared-memory layer.

5.4 Key Points

The macrocell structure eliminates a lot of unnecessary requests for remote data. It allows DIRT to scale reasonably well on small to medium clusters. However, end-to-end round trip message latency still significantly limits the efficiency and scalability of DIRT in medium and large cluster configurations. DeMarle measured end-to-end latency for remote brick access to be about 19us on 32 nodes of the same Dell cluster described in Section 4.6.This large latency limits the scalability of DIRT.

When a worker node requests bricks from a remote node, there is a trade-off between the amount of network bandwidth and message system overhead that DeMarle considered. Fetching a single brick on demand results in using only the necessary global bandwidth and would make best use of the remote memory cache. However, as each request has some message overhead and latency, this results in lower overall efficiency. Instead, DIRT precomputes the set of bricks that a ray may intersect with in a particular macrocell each time there is a miss and requests them all in one message. While this results in lower overall message overhead and lost time due to message latency, it wastes network bandwidth and unnecessarily pollutes the local brick cache with data that in some cases is ultimately not needed.

In addition to resulting in a nontrivial amount of extra computation to compose a set of brick requests into a single request, this precomputation step required source code changes and added software complexity to the core of the ray tracer. The additional complexity could have been avoided if the underlying message passing system were capable of delivering requests at a sufficiently low latency and with sufficiently low overhead.

DIRT is an interesting application for the ULN architecture in that it has one to several compute threads and a communication thread per node and it relies on frequent unpredictable messages. When a computation thread needs to communicate, the CPU can switch to other tasks. The communicating thread is blocked until a round-trip message occurs. The remote nodes involved in a communication are also simultaneously computing pixel values. The act of responding to requests induces additional local overhead. If occasional polling is used, the effective round-trip latency is negatively affected. If frequent polling is used there is a high induced overhead. If interrupts are used for notification, an interrupt penalty

is incurred for each message. Given the application's small, frequent messages, the total notification overhead can be significant.

DIRT is an excellent test case for the architecture as DIRT's computation and communication architecture match the hardware architecture well. The concept and use of communication and computational threads is very similar in both models. Thus, DIRT represents an interesting test of the architecture's capabilities. DIRT's `communicator` thread can be executed as a thread on the SMT processor and can take full advantage of both the interthread communications primitive as well as the message notification primitives that this architecture provides. One or more `Renderer` threads can be mapped on to the remaining compute threads on the SMT processor.

The low overhead, low latency characteristics of the ULN architecture reduces the granularity of remote data access that is necessary to get efficient scaling to any number of nodes when compared to the commodity interconnect cluster architecture that DIRT originally targeted. Having such an efficient architecture may have also reduced or eliminated the need to make as many changes to the RTRT code base that were required to precompute remote brick intersections and coalesce remote block prefetches. Reducing the need for such aggressive prefetching has the benefit of minimizing wasted bandwidth and lowering cache pollution. ULN's benefit on DIRT is shown in Chapter 8.

CHAPTER 6

DIRT CHARACTERIZATION

This chapter presents performance data gathered from running DIRT on the Dell cluster described in Section 4.6.This performance data is analyzed to gain an understanding of the time spent in computation versus time spent in the communication system. The code was characterized with the help of the TAU utilities as well as internal instrumentation. The performance numbers presented in this section represent the timings with these overheads removed according to the results and methods in Sections 4.6 and 4.7.

6.1 Experimental Setup

While DIRT is a complete ray-tracing system capable of rendering many types of scene models, the runs examined here are of a visualization of isosurfaces within a large, generated, regular 3-D grid volume dataset. The volume dimensions are 2,892 by 2,892 by 2,892 data points, and each data point is a a 2-byte "`short`" value. The total volume is approximately 48 GB. The volume data is distributed evenly amongst the workers such that on a 32 worker node run, each node has 1.5 GB of scene data. As each of the nodes has 2 GB of memory, the remaining 512 MB per node is shared by the code, the local variables, and by the local software shared memory cache that caches remote scene data.

The volume consists of 20 randomly positioned spheres. The center of each sphere is a randomly chosen isovalue within the range of the signed `short`. The isovalue of each

sphere decreases linearly along the radius of the sphere. Four sample images of this scene are shown in Figure 21. These images are taken from a sequence 5 frames apart. In each frame, the camera rotates clockwise around the scene by 5 degrees and the isovalue being viewed increases by 1,000. Thus the scene appears to rotate counter clockwise about 20 degrees in each of these 4 images as the spheres appear to shrink. It should be noted that the apparent holes in the spheres are manifestations of the edges of the volume.

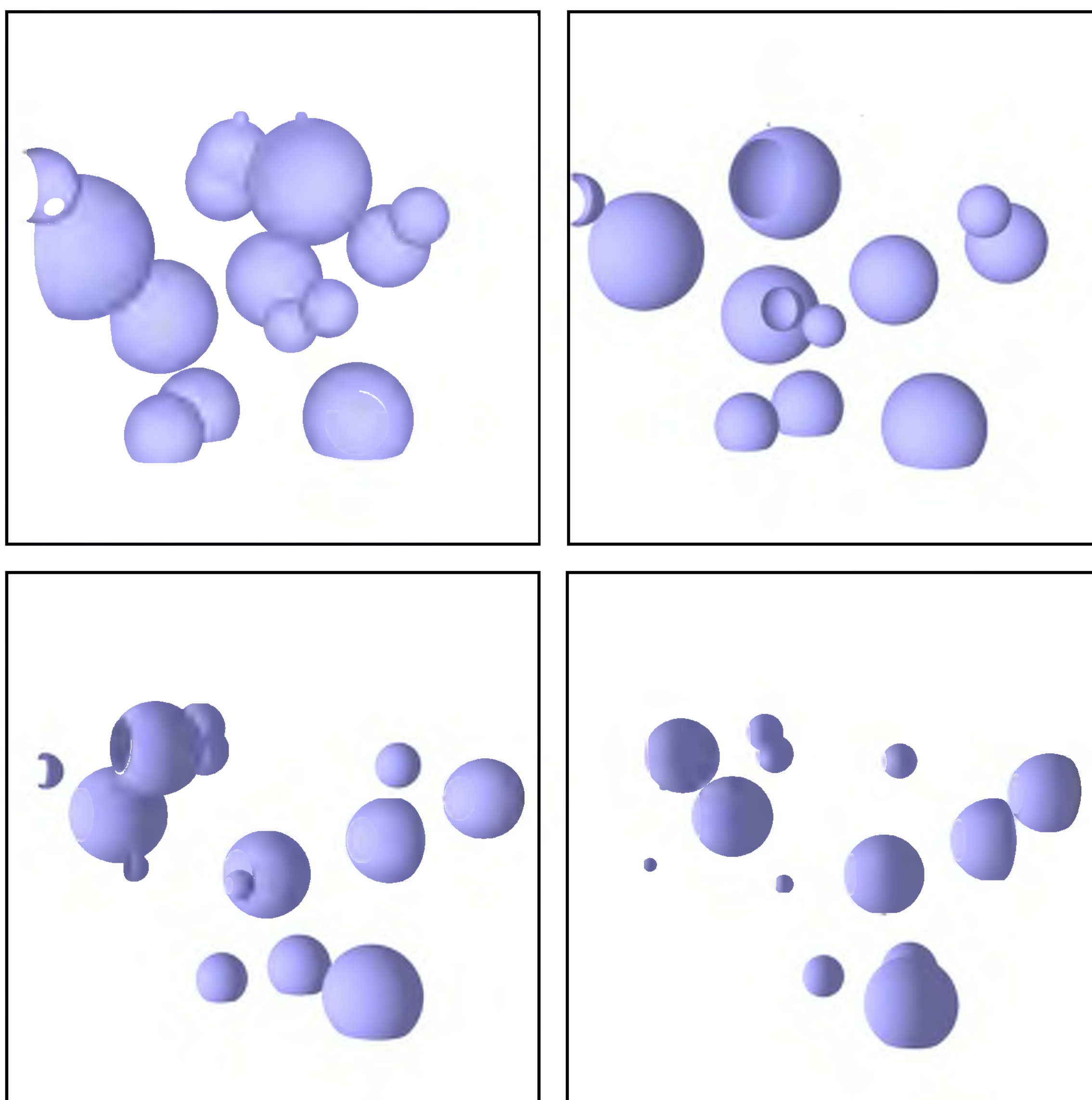

Figure 21: Sample images from DIRT run

Several runs were made using a total of 33, 41, 49, 57, and 65 nodes. In DIRT, 1 of the nodes is always a supervisor node, so there are 32, 40, 48, 56, and 64 worker nodes in each of these runs, respectively. An additional variable in the experiment is to vary the number of `Renderer` threads per worker node from 1 through 4.

Throughout this section, these runs are identified using the nomenclature of *X*w_*Y*r, where *X* is the number of worker nodes in the job and where *Y* is the number of `Renderer` threads per node. For instance, 32w_1r refers to the run that has 32 worker nodes, or 33 total nodes including the supervisor node, and 1 `Renderer` thread per worker, or 3 threads total per worker node including the `main` thread and the `communicator` thread.

Sections 6.2.1 through 6.2.7 and Sections 6.3.1 through 6.3.3 show abstracted code for the ray-tracers key components. This code is shown to illustrate the operation of DIRT and to provide an explanation of the performance analysis of the code presented in Section 6.3.5 and Section 6.4. These code in these sections is annotated to illustrate where time is spent in various parts of the code for a single node, specifically node 1, on a single 32w_1r experiment. While the analysis used in Chapter 8 uses the average information across all of the nodes in this experiment, examining the single node provides a concrete view of the analysis as well as the operation of DIRT. All of the times annotated in the code in these sections are in seconds. These annotated figures in Sections 6.2 and 6.3 show the call-tree hierarchy and delineate the time spent in initialization, computation, communication, and load imbalance. Sections 6.2.1 through 6.2.7 illustrate the `Renderer` thread call-tree hierarchy, while Sections 6.3.1 through 6.3.3 illustrate the `communicator` thread call-tree hierarchy.

The timing annotations in these figures follow the TAU naming conventions. The inclusive time of a function is annotated in the figure using the abbreviation "`i`." The exclusive time of a function is annotated in the figure using the abbreviation "`e`." As a reminder, the inclusive time is the time spent executing in and under a function including all child function calls, where the exclusive time is the time spent only in the function proper. Each figure shows one function detailing the inclusive and exclusive time spent in that function, that is annotated on the top line of the figure with the function declaration. Each child function call is also annotated with the child's inclusive time. The inclusive time for a parent function, or "`i`" time, is equal to that function's exclusive time, or "`e`" time, plus the inclusive time for all of the child functions called. Mathematically this can be expressed as:

$$i_{parent} = e_{parent} + i_{child_1} + i_{child_2} + \ldots + i_{child_n}$$

Since the TAU tools instrument the source code, built-in library calls are not measured separately. Instead their time is included as part of the time spent in the calling function. In a few instances a routine is called from multiple functions. Since the particular instrumentation of DIRT used did not explicitly separate out the time spent in routines based on where they were called from, it is necessary in a few places to compute the times spent in each invocation based on the unaccounted time in the calling routine. In the single thread case, it is possible to account for the time spent in all of the calls exactly. In the multiple thread runs, there are a few places where it is not possible to know exactly how the time is split between a few invocations of a routine. However, the exact split is not material to understanding how the time was spent between compute and communication as the uncertainty is only whether the time is spent in communication in one part of the code or an-

other. The multiple thread case will be described later. In all cases, the sum of the computed times across all such invocations was checked against the total time spent in these routines.

6.2 `Renderer` Thread Characterization

The figures in Sections 6.2.1 through 6.2.7 show the **`Renderer`** thread's call tree hierarchy on the worker nodes annotated with the corrected TAU timing numbers from node 1 of the 32t_1r run. The precision shown in the figures and discussed in the text represents the full 1 μs granularity precision reported by TAU. Though that leads to more precision than is necessary, the full precision is reported to illustrate that the times of the children functions and the parent exclusive time add up to the parent inclusive time.

6.2.1 `Renderer::run` Function

Figure 22 shows the annotated body `Renderer` thread, including the main loop. The total time spent in the `Renderer` thread is 185.925720 seconds. Of that, a negligible 12 μs is spent in `PerProcessorContext`, `~PerProcessorContext`, and `sock` performing initialization tasks. Since this time occurs once per invocation of the ray-tracer, this overhead is subtracted out of the total time of the operation of the ray-tracer's main loop, leaving 185.925708 seconds as the key time for that loop. The time spent in `Context` is time spent in per-assignment data-structure setup. Both `makeRay` and `pack_pixel` are per-pixel operations. The ray direction is computed in `makeRay`. The pixels are packed in preparation for sending in `pack_pixel`. The `Renderer::run` exclusive time as well as the `Context`, `makeRay` and `pack_pixel` inclusive times are all local computation.

```
Renderer::run() { 185.925720 i, 0.401760 e
  PerProcessorContext() 0.000007 i
  ~PerProcessorContext() 0.000002 i
[...] // initialization
  sock() 0.000003 i
  while (true) {
[...]
    Context(); 0.000881 i
[...]
    popTask(); 27.669846 i
[...]
    //go through the assigned assignments
    For each tile assignment {
[...]
      //finally ray trace the tile
      for each pixel {
        makeRay(); 0.352950 i
        traceRay(); 157.176001 i
[...]
        pack_pixel(); 0.169314 i
      }
[...]
      send_buffs() 0.154946 i
    }
  }
}
```

Figure 22: Annotated `Renderer` thread `run` function (main loop)

The `Renderer` thread calls the `popTask` routine to receive new work assignments. The time spent in this function is a mixture of communication, computation, and load imbalance and is shown in Section 6.2.2. The traceRay routine is where rays are traced through the scene data. This time is split between computation and communication. Details are shown in Sections 6.2.3 through 6.2.7. The pixels are sent to the supervisor in `send_buffs`. This routine is a thin thread-safe wrapper around a send call and is all communication time. Finally, the exclusive time spent in the `run` routine is due to local per-assignment and per-pixel computation.

6.2.2 `popTask` Function

The **`popTask`** routine, shown in Figure 23, is a semaphore wrapper that waits for a task to be queued from the supervisor node. In **`popTask`**, the **`tasksemavail`** semaphore is used to wait for work to be placed in the local work queue. When the **`communicator`** thread receives work, it places the work in this queue and signals "**`up`**" on this semaphore releasing the **`tasksemavail->down`** call. There is an additional **`tasksemaccess`** semaphore that is used to protect the critical section of removing the tasks from the local shared work queue. This semaphore is needed when there are multiple renderer threads per node to ensure that each renderer thread atomically removes a task from the queue.

The time spent in **`tasksemavail->down`** is a combination of communication overhead and load imbalance time, as this is where the **`Renderer`** thread waits for the next frame to start at the end of each frame. It is not possible to know explicitly how much of the time is spent in communication versus load imbalance since the communication overhead appears to the endpoint as a component of the load imbalance. Therefore, the time spent in communication here is estimated by averaging the amount of time spent in communication per cache miss times the number of assignments. The average time for these requests in this instance is about 426 μs per communication. In this case, a total of about

```
TaskManager::popTask() { 27.669846 i; 0.018432 e
[...]
  tasksemavail->down(); 27.633445 i
[...]
  tasksemaccess->down();
[...] // remove the task from the work queue
  tasksemaccess->up(); 0.017969 i
[...]
}
```

Figure 23: Annotated `Renderer` thread `popTask` function

6.177874 of the 27.633445 seconds spent in the `tasksemavail->down` routine are attributed to communication. The remaining 21.455571 seconds are due to load imbalance. The minimal `popTask` exclusive time is local computation time and the `tasksemaccess->up` is local communication time.

6.2.3 `traceRay` Function

The `traceRay` routine, shown in Figure 24, calls the `intersect` routine, where the ray is checked to see if it intersects with the dataset, and then colors the resulting pixel accordingly. If the ray hit the scene, it is colored with `shade`, otherwise, it is colored with `get_bgcolor`. The `traceRay` exclusive time, the `shade` inclusive time and the `get_bgcolor` inclusive time are purely local computation. The `intersect` routine is a mixture of communication and computation, broken down further in Section 6.2.4.

6.2.4 `intersect` Function

The `intersect` routine, shown in Figure 25, performs one of the key computations in a ray tracing system - determining the closest point for which the ray intersects a particular surface. This version, which is specialized for isosurfaces of 3-D volumes, uses a macrocell hierarchy to accelerate the query. After setting up the parameters for the mac-

```
Renderer::traceRay() { 157.176001 i; 0.390552 e
[...]
  intersect(); 156.622203 i
  if(hit) {
    // color the object according to the scene data
    shade(); 0.130357 i
  } else {
    // no scene data found, thus we hit the background
    get_bgcolor(); 0.032899 i
  }
}
```

Figure 24: Annotated `Renderer` thread `traceRay` function

```
DISOVolume::intersect() { 156.622203 i; 1.205502 e
[...]
    getVolumeDpy(); 0.043382 i
[...] // calculate where ray crosses macrocell boundaries
    isect(mc_depth-1, ...); 155.373319 i
}
```

Figure 25: Annotated `Renderer` thread `intersect` function

rocell hierarchy walk, `intersect` calls `isect`, the routine that walks the macrocell hierarchy and, if necessary traverses the scene data. The `intersect` exclusive and `getVolumeDpy` inclusive times are local computation. The `isect` time is split between communication and computation.

6.2.5 `isect` Function

The `isect` routine shown in Figure 26 has a more complex call hierarchy than the previous routines. In addition to being called from `intersect`, the `isect` routine also recursively calls itself to traverse the macrocell hierarchy. If the routine reaches the bottom level of the macrocell hierarchy, it performs an intersection test on the underlying scene data.

The `getrhos_many` routine gathers the set scene data values associated the ray may intersect with. The `HitCell` routine solves a cubic function to determine if the ray hits the cell, and if there is a hit, `GradientCell` computes the angle of intersection. The `getrhos_many` routine is a mixture of computation and communication. The `isect` exclusive time, the `HitCell` time, and the `GradientCell` time are all computation.

6.2.6 `getrhos_many` Function

The `isect` function produces a list of cell corners needed to trace the ray through the bottom level of the macrocell hierarchy. The `getrhos_many` routine, shown in Fig-

```
DISOVolume::isect() { 155.373319 i; 4.972253 e
[...]
    if at bottom level {
[...]
      //pretraverse to find which cells are needed
      //make as few calls as possible to get data
      getrhos_many(); 149.679501 i
[...]
      //now traverse the data
      for each data point {
[...]
        if in cell bounds { //may hit, solve explicitly
[...]
          if (HitCell()) { 0.682755 i
            if (hit) {
[...]
              GradientCell(); 0.038810 i
[...]
              break;
            }
          }
        }
      }
    } else {
     //not yet at bottom mcell level, traverse mcells in
     //this level. if we hit any possible bricks, go down to
     //next level
      for each cell at this level {
[...]
        if can hit in this mcell {
[...]
          isect(mc_depth-1, ...); //time already accounted
        }
[...]
      }
    }
}
```

Figure 26: Annotated `Renderer` thread `isect` function

ure 27, determines the set of cachelines needed to traverse the ray and requests each cacheline in sequence by calling the `get_data` function on each line. The `get_data` routine locks the cacheline from being evicted from the cache and returns a pointer to the data. The `getrhos_many` routine then retrieves the needed value from each cacheline and releases the line with the `release_data` call allowing it to be evicted from the cache if necessary.

```
getrhos_many() { 149.679501 i; 6.908229 e
[...]
  for () {
    //get the next key
    get_data(); 140.736236 i
    //get all data values that we need out of that key
[...]
    //release the key
    release_data(); 2.035036 i
  }
}
```

Figure 27: Annotated `Renderer` thread `getrhos_many` function

6.2.7 `get_data` Function

The last significant routine in the `Renderer` thread call hierarchy is the `get_data` routine, shown in Figure 28. This routine is a member of the dataserver class where access to the shared memory data is managed. It does the check to see if the requested data is locally owned, remote but cached, or remote and not presently cached and completes the appropriate action to satisfy the data request. If remote data is requested, this thread blocks until the data is returned to the `communicator` thread running on this node and until the `communicator` thread notifies this routine that the data has arrived via a semaphore.

The exclusive time spent in the `get_data` routine as well as the inclusive time spent in the `get_access`, `release_access`, `get_sole_access`, `release_sole_access`, and `release_sole_access_but_retain` routines are all a part of the software shared memory system. These pieces are considered part of the local computation.

The time spent in the `send_msg` and `up` and `down` calls is time spent in the communication system. In particular, the time spent in the `send_msg` call is overhead associated with sending a message to the home node for the shared memory request. The total

```
dataserver_direct::get_data() { 140.736236 i; 1.811214 e
[...]
  if (own == myrank) {
[...] // satisfy locally
  } else {
[...]
    semaphore.get_access(); 2.140364 i
    if (cache hit) {
[...]
    } else { // miss
      semaphore.release_access(); 0.214406 i
      semaphore.get_sole_access(); 0.080460 i
[...]
      semaphore.release_sole_access(); 0.198639 i
[...]
      send_msg(); 3.481132 i
[...]
      // wait for the data response
      cache[pos]->down(); 132.520182 i
     |while (cache[pos]->key != key) { // not my response
     |  cache[pos]->sem2->up();    // let someone else try
     |  cache[pos]->sem2->down();  // sleep and try again
     |}
[...]
      release_sole_access_but_retain();  0.289839 i
[...]
    }
  }
}
```

Figure 28: Annotated `Renderer` thread `get_data` function

time spent in the **down** and **up** calls is the time spent waiting for the return of data. This wait time includes message latency in both directions as well as message handling time on the remote node. Unlike **popTask()**, there is not a component of load imbalance in this time. The message handling time on the remote node is the subject of Section 6.3.

The loop marked with the bar in Figure 28 is due to the way the shared memory cache is implemented to support multiple **Renderer** threads per node. When a cache miss occurs, the **get_data** routine waits on a semaphore associated with the cacheline where the miss data will be stored. When multiple **Renderer** threads are running, two or more threads could be waiting for two different cachelines that map to the same cache index.

Since the semaphore is tied to the index, not to a unique cacheline, the wrong thread may see the "`up`" on the semaphore. If so, that thread resignals "`up`" and then waits again for the intended reference. In the single renderer thread per node case, this loop is never executed as there are no simultaneous conflicts to the same cacheline.

Unfortunately, in the multithreaded case, these iterations are common, thus the marked loop is not very optimal. The large number of loop iterations is caused when the target thread does not see the reference immediately and the interfering thread repeatedly releases and reacquires the semaphore, leading to a lot of unnecessary overhead. At a minimum, this thread should call a `yield` function in between the `up` and the `down` calls in the `while` loop to give the target thread the opportunity to be scheduled in the case that the processors on the node are over-subscribed. The current implementation can lock the intended node out of getting the intended response for some unbounded amount of time. A more robust solution would be to have a semaphore associated with each thread and then to tag requests and responses to wake up the exact thread that was waiting for the request. However, the primary focus of this work is to examine an existing code.

This is the key area where it is impossible to tell exactly how much time is spent in some parts of a function. Since `up` and `down` are also used to implement `popTask` we cannot deduce exactly how much of the `get_data` inclusive time is spent in `up` versus `down`. However, it is easy to tell amount of time spent in the sum of the two. In the end it all counts as time waiting for a cache miss response. Since we are interested in understanding the time spent in the communication subsystem, in the multiprocessor analysis in Section 6.4, the time is aggregated into a single wait time. Since this loop is never entered in the single-thread case, our analysis of where the time is spent in the code is simplified.

6.2.8 `Renderer` Thread Summary

Table 6 shows the average time, standard deviation of that time, and the invocation count for each of the above functions in the `Renderer` thread. For each function, it is noted if the time listed is the inclusive time or the exclusive time. The exclusive times are listed for all of the functions that are broken down above. For leaf functions in the call-tree or for functions where the hierarchy is not broken down above, the inclusive time is listed.

Each of the nodes has roughly the same total time, due to the fact that each of the nodes synchronize at each frame step as well as at the end of the simulation. There is a slight load imbalance between the nodes, partially evidenced by the somewhat larger standard deviation of the `popTask` routine. The time spent in each of the compute components above is shown graphically in Figure 29. The time spent in each of communicate and load balance components is shown in Figure 30.

6.2.9 `Renderer` Thread Analysis

For the scene parameters described on the aforementioned cluster most of the renderer thread time is spent waiting for remote data due to scene cache misses. The summary of the total time spent in computation versus communication versus load imbalance is shown in Figure 31.

Since there is no little to no TAU measurement overhead in the load balance portions of the measurements, the time lost in the load balance overhead, as the remaining time components reduce, the fraction of the time in load imbalance appears larger in Figure 31 than it would be if there were no measurement overhead. To get a better estimate of the expected load imbalance if measurement overheads were not present, the time spent in load imbalance is scaled by a factor of the total corrected `Renderer` thread runtime divided by

Table 6: Summary of `Renderer` thread key function times across all nodes

Function	Average Time (s)	Standard Deviation	Invocations	Category
Renderer::run excl	0.639	0.162	1	compute
Context incl	0.000512	0.000116	14081	compute
popTask excl	0.0170	0.00193	14081	compute
tasksemavail->down incl	32.6	2.55	14081	load balance
tasksemaccess->down incl	0.00681	0.000478	14081	compute
tasksemaccess->up incl	0.0169	0.000710	14081	compute
makeRay incl	0.300	0.0188	3604480	compute
traceRay excl	0.304	0.0447	3604480	compute
intersect excl	1.15	0.0570	3604480	compute
getvolumeDpy incl	0.0269	0.0108	3051845	compute
isect excl	4.65	0.163	5793250	compute
getrhos_many excl	6.57	0.240	992353	compute
get_data excl	1.58	0.0842	2663267	compute
semaphore.get_access incl	1.99	0.0731	2579696	compute
semaphore.release_access incl	0.197	0.00565	289719	compute
semaphore.get_sole_access incl	0.0742	0.0104	289719	compute
semaphore.release_sole_access incl	0.182	0.00683	289719	compute
send_msg incl	4.57	0.728	289719	comm
cache[pos]->down incl	130	1.85	289719	comm
release_sole_access_but_retain incl	0.258	0.00946	289719	compute
release_data incl	1.84	0.0802	2663267	compute
HitCell incl	0.602	0.0422	871690	compute
GradientCell incl	0.0355	0.00385	341814	compute
shade incl	0.110	0.0174	341814	compute
get_bgcolor incl	0.0162	0.0102	3262666	compute
pack_pixel incl	0.151	0.0154	3604480	compute
send_buffs incl	0.392	0.143	14080	comm

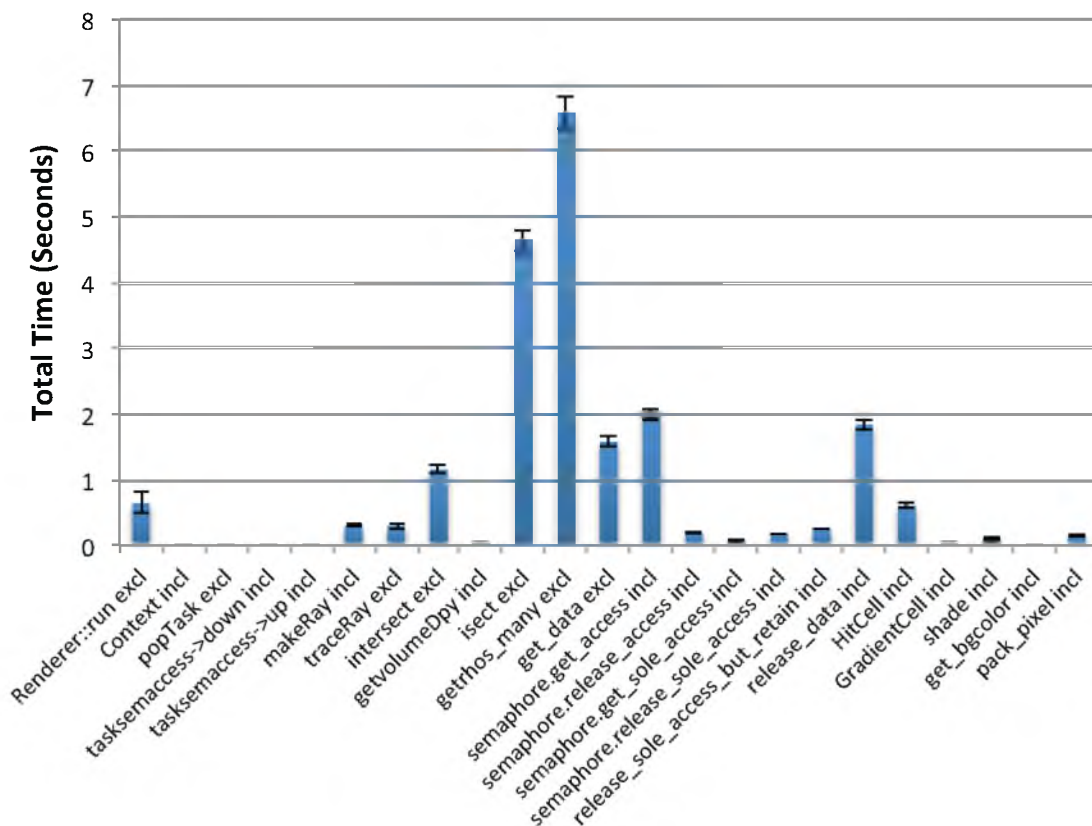

Figure 29: `Renderer` compute components

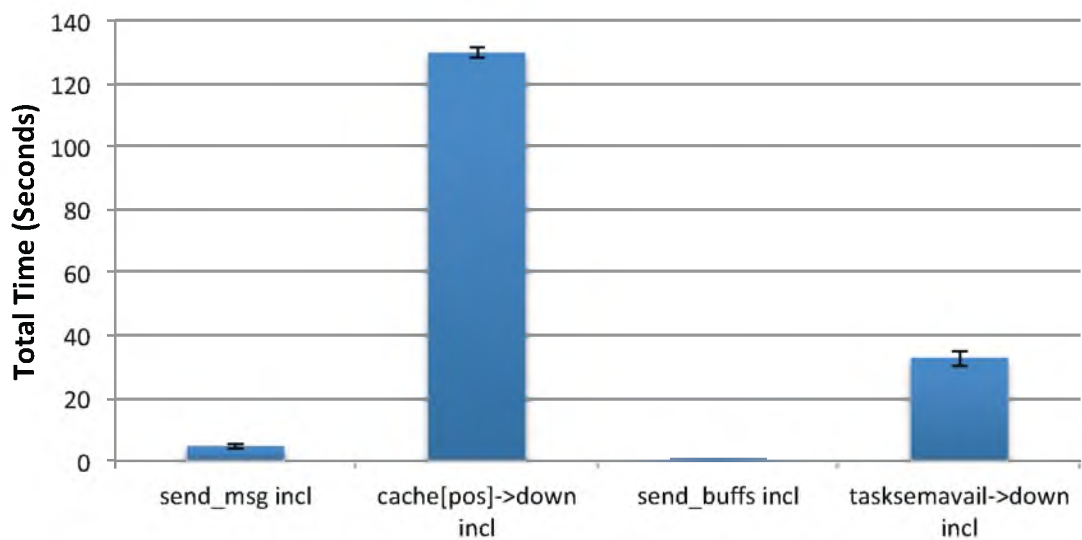

Figure 30: `Renderer` communication and load balance components

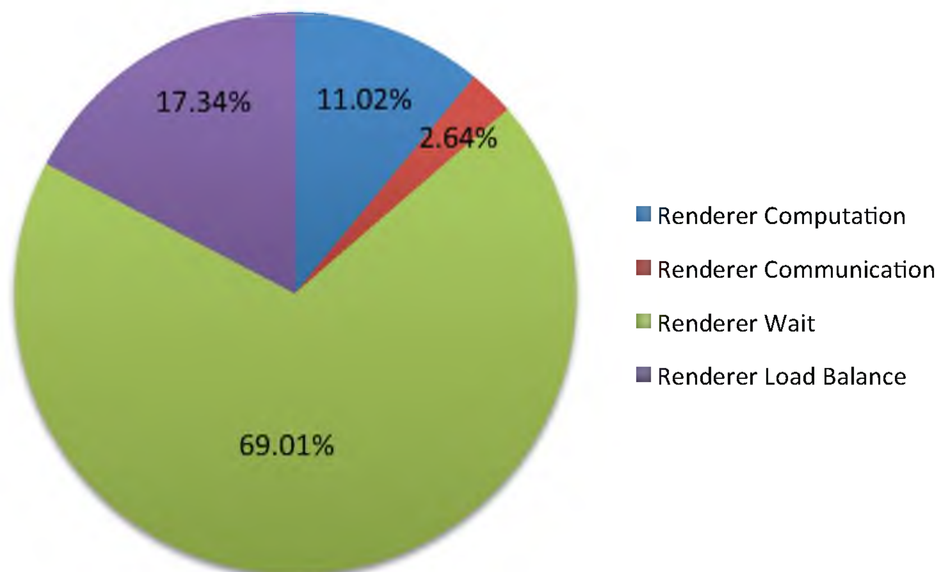

Figure 31: `Renderer` thread breakdown summary

the uncorrected `Renderer` thread runtime. The corrected load imbalance time is estimated at 24.992601 seconds and the resulting breakdown is shown in Figure 32.

On this cluster on this particular run, the average time per miss is 464 μs. The message request is 20 user bytes and the response is 3,472 user bytes. The exact amount of time spent in the network versus in the OS is not visible from this run. Given the gigabit Ethernet network, assuming full bandwidth, the serialization of this data only accounts for 28 μs, a small fraction of that time. The cluster is built using a hierarchy of switches that has a smaller bisection than needed to support all of the nodes at full bandwidth. It is likely that the serialization latency here and at each hop accounts for significantly more than that 28 μs; however, there is clearly additional OS overhead associated with the messaging system on the nodes.

Only the compute times and the counts of communications events from this analysis are used in the model in Chapter 8. Thus, that model is independent of the particular load imbalance overheads and communications overheads analyzed here.

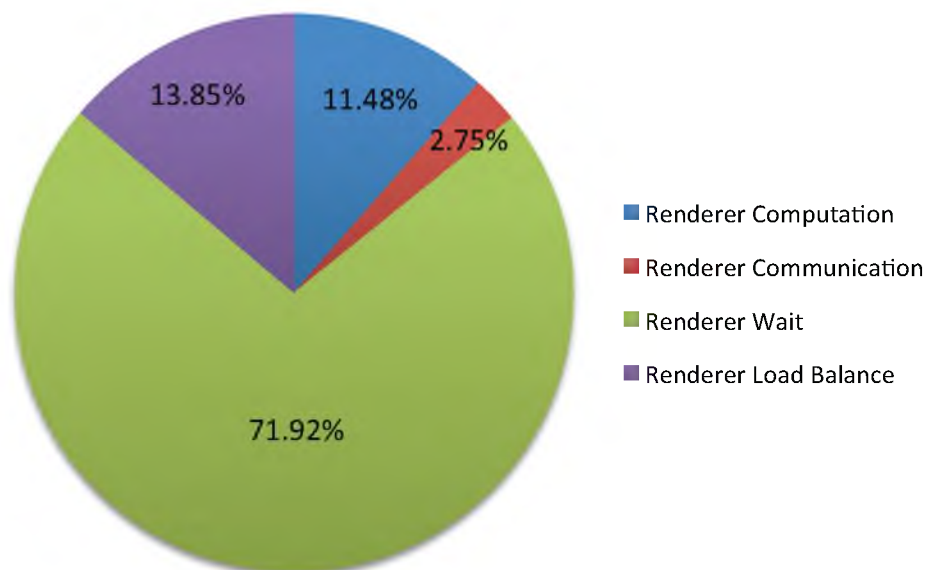

Figure 32: Corrected `Renderer` thread breakdown summary

The total `Renderer` compute time per node spent per frame is 0.186656 seconds, per assignment it is 1.472 ms, and per cache miss it is 71.514 μs. Technically this is composed of an average of a 13.662 μs overhead per assignment plus a 70.850 μs average time between misses within an assignment, however, at the macroscopic level the critical number for the model is the average total time between cache miss requests. These key values are summarized in Table 7.

Table 7: Summary of key `Renderer` thread characteristics

Characteristic	Value
Total compute time per frame (all nodes)	6.03 s
Average compute time per node per frame	0.188 s
Average compute time per assignment	1.47 ms
Average compute time per cache misses	71.5 μs
Total assignment per frame (all nodes)	4096
Average assignments per node per frame	128
Average cache misses per node per frame	2630
Miss Request Size	20 bytes
Average Pixel Assignment Send Size	796 bytes

6.3 Communicator Thread Characterization

In addition to one or more `Renderer` threads, each node has a `communicator` thread that handles cache misses requests on behalf of other nodes and receives work assignments from the supervisor node. The following figures show this `communicator` thread and the associated `dataserver` call tree hierarchy annotated with the corrected TAU performance data from node 1 of the 32w_1r run.

6.3.1 `communicator::run` Function

Figure 33 shows the thread's main function. This implementation of the `communicator` thread is built using Unix sockets for communication. After a short initialization, this thread sits in a loop waiting for a message to arrive. When a message arrives, the header is received and the appropriate handler is called. Since `select` is a library function, it is not instrumented by TAU and is not explicitly timed, hence the exact amount of time spent in the select call is unknown. However, most of the exclusive time in `communicator::run` is spent in the `select` call.

The computed time for the `communicator` thread is much higher than the computed total time of the `Renderer` thread above. This is primarily due to the fact that the TAU instrumentation overhead is much smaller in the `communicator` thread due to the relatively fewer dynamic subroutine call invocations. It is also due in small part because the `communicator` thread is started before the `Renderer` threads. The exact time the thread runs is unimportant. What is noteworthy is that the `communicator::run` thread is idle, waiting for requests to arrive, approximately 93.6% of the time, and is generally available to service requests as soon as they arrive. Finally, what is important is the amount of time spent handling each of the request types.

```
//this is the communicator thread that runs in the background,
//waiting for incoming messages, and directing them to
//registered methods when they come.
communicator::run() { 242.482501 i; 227.031638 e
[...]
  while(true) {
[...]
    select();
[...]
    recv();  // get header
[...]
    switch() { // figure out which handler to call - invoke one of:
      case DSM_MESSAGE: dataserver_group::handlemessage(); 15.272459 i
      case NEW_TASK: TaskManager::handlemessage(); 0.167519 i
      case SCENE_UPDATE: ViewManager::handlemessage(); 0.010339 i
      case BARRIER: barrier_group::handlemessage(); 0.000546 i
    }
  }
}
```

Figure 33: Annotated `communicator` thread main function

The `TaskManager::handlemessage`, `ViewManager::handlemessage`, and `barrier_group::handlemessage`, routines comprise well under 1% of the total runtime and are not broken down in detail here. The `barrier_group` receives a few messages during initialization and at the very end of the program as the ray-tracer exits, hence the time spent in that routine is initialization overhead. The `ViewManager` receives a single message with the new view and scene parameters at the start of each frame and the `TaskManager` receives a number of task assignments for each frame. While the time spent in these two handlers is small, they are an important part of the ray-tracer's operation. Finally, the software shared memory messages are handled by the `dataserver_group`. This is where the `communicator` thread spends the majority of its nonidle time.

6.3.2 `dataserver_group::handlemessage` Function

Figure 34 shows the `dataserver_group::handlemessage` routine. Since there are multiple implementations of the dataserver, the `dataserver_group` is a class hierarchy that encapsulates the set of dataserver types. In this case it calls the dataserver handler associated with the direct mapped cache implementation of the software shared memory system. The `recv_buff` call receives the dataserver packet, which may be either a shared memory miss request or a data return from a previous request.

6.3.3 `dataserver_direct::handlemessage` Function

The `dataserver_direct::handlemessage()` routine is shown in Figure 35. A large fraction of the time is spent in the message passing system, rather than actually servicing the request. For a received request, the time to get the shared memory data for the response is a part of the exclusive time of this routine. The exclusive time also comprises the time required to look up the relevant cache miss information to determine where to place the received data. It is useful to note that the `up` call in this routine releases the `tasksemavail` semaphore in Figure 28.

```
//handle incoming dataserver messages from remote nodes
dataserver_group::handlemessage(){15.272459 i; 0.450487 e
[...]
{
[...]
  recv_buff(); 2.079615 i (computed)
[...]
  dataserver_direct::handlemessage(); 12.742357 i
}
```

Figure 34: Annotated `communicator` thread `dataserver_group::handlemessage` function

```
dataserver_direct::handlemessage(){12.742357 i;0.583733 e
[...]
  //get header
  recv_buff(); 4.366069 i (computed)
  switch (MSGID) {
  case L_DM_RQST_DATA:
[...] // find the the data to be returned
    send_msg(); 6.232386 i
    break;
  case L_DM_SENT_DATA:
[...]
    semaphore.get_sole_access(); 0.144310 i
    recv_buff(); // (time combined with first invocation above)
[...]
    up();  // data arrived, release worker 1.415859 i
    break;
  }
}
```

Figure 35: Annotated `communicator` thread `dataserver_direct::handlemessage` function

6.3.4 `communicator` Summary

Table 8 shows the average time, standard deviation of that time, and the invocation count for the important functions in the `communicator` thread. For each function, it is noted if the time listed is the inclusive time or the exclusive time. The exclusive times are listed for all of the functions that are broken down above. For leaf functions in the call-tree or for functions where the hierarchy is not broken down above the inclusive time is listed.

Due to the instrumentation of DIRT, it is not possible to tell the amount of time spent in the **`recv()`** call in `communicator::run`. However, the receive call is similar in the amount of data read to the `dataserver_group` **`recv_buff()`** call implying an estimated 3.35 μs per call. It is invoked once for each call to **`dataserver_group::handlemessage`**, **`TaskManager::handlemessage`**, and **`ViewManager::handlemessage`**, for an estimated total **`recv()`** time of 1.99 seconds.

Table 8: Summary of `communicator` thread key function times across all nodes

Function	Average Time (s)	Standard Deviation	Invocations	Category
dataserver_group::handlemessage excl	0.381	0.0258	579438	compute
dataserver_group recv_buff incl	1.94	0.0396	579438	comm
dataserver_direct::handlemessage excl	0.573	0.0299	579438	compute
dataserver_direct recv_buff incl	4.09	0.0830	869158	comm
dataserver_direct send_msg incl	12.9	3.40	289719	comm
semaphore.get_sole_access incl	0.131	0.0152	289719	compute
cache[pos]->sem2->up incl	2.65	2.59	289719	local sync
TaskManager::handlemessage excl	0.0315	0.00131	12430	compute
TaskState::getAssignment excl	0.00776	0.000432	12430	compute
TaskState recv_buff incl	0.0521	0.00182	12430	comm
pushTask excl	0.0311	0.00204	12430	compute
tasksemaccess->down incl	0.00601	0.000422	12430	compute
tasksemavail->up incl	0.0169	0.000782	14080	local sync
tasksemaccess->up incl	0.0149	0.000691	12430	compute
ViewManager::handlemessage excl	0.00194	0.00194	113	compute
getframe excl	0.000754	0.000057	111	compute
framestate.get_state excl	0.000193	0.000033	111	compute
get_state recv_buff incl	0.000732	0.000015	222	comm
setVolumeDpy	0.000043	0.000008	111	compute
animate incl	0.000023	0.000008	111	compute
camera->set_eye incl	0.000031	0.000005	111	compute
camera->set_lookat incl	0.000035	0.000006	111	compute
camera->set_fov incl	0.000025	0.000007	111	compute
camera->set_up incl	0.000025	0.000006	111	compute
camera->setup incl	0.000155	0.000007	111	compute
Thread::currentSeconds incl	0.000086	0.000006	111	compute
taskm->endframe incl	0.00304	0.000114	111	compute

6.3.5 32w_1r Communicator Thread Analysis

The `communicator` thread spends only 0.63% of its time in computation associated with handling misses. The bulk of the time, 87.86% is spent idle waiting for incoming messages. The idle time here is the total `Renderer` time minus the time spent running in the `communicator` thread. The bulk of the time spent while the `communicator` thread is running is in messaging overhead tasks with 10.09% of the time spent in sends and receives, and 1.42% of the time spent in operations associated with waking the `Renderer` thread. This breakdown is shown in Figure 36.

6.3.6 Combined 32w_1r Analysis

Combining the `communicator` communication time with the `Renderer` communication time, indicates that 14.11% of the total runtime is spent in the user-level communication system. This is in contrast to only 11.60% of the time spent in actual computation between the two threads. Clearly, even ignoring the invisible OS overheads

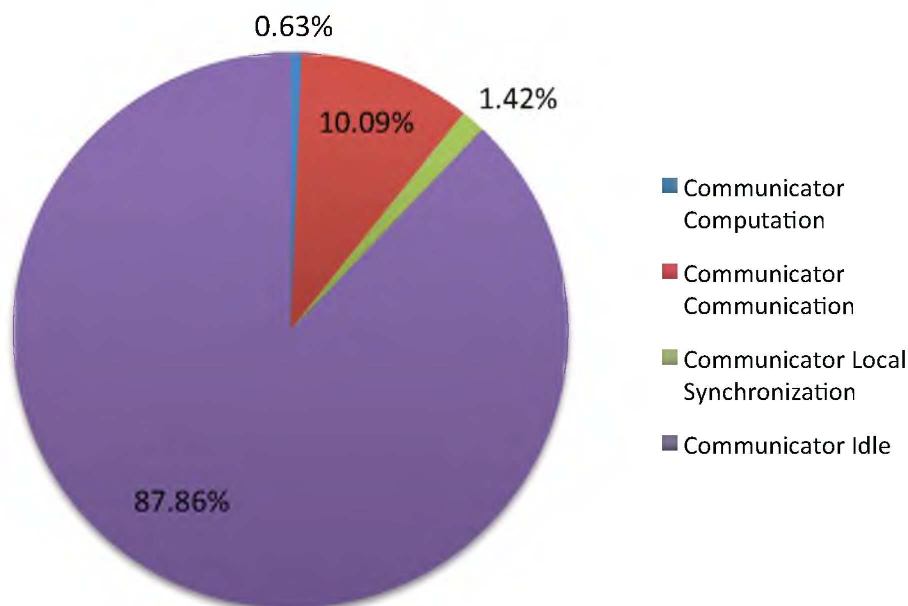

Figure 36: `communicator` thread breakdown summary

associated with messaging, there is plenty of room for optimization of the communication system. This summary is shown in Figure 37.

The summary with the corrected load balance overhead is shown in Figure 38.

Table 9 shows the key characteristics of the `communicator` threads for the 32w_1r run.

6.4 All DIRT Runs

As a part of the experimentation, the number of nodes and number of `Renderer` threads per node. Figure 39 shows an increase in performance as we scale from 1 to 32 nodes. Figure 40 shows the speed up of the components of each of those runs as the number of nodes scale. The computation time scales perfectly with the number of nodes. The communication and wait times appear to scale super-linearly from 32 to 40 nodes and then scale close to linearly thereafter. The primary reason for this is that these runs were made on a

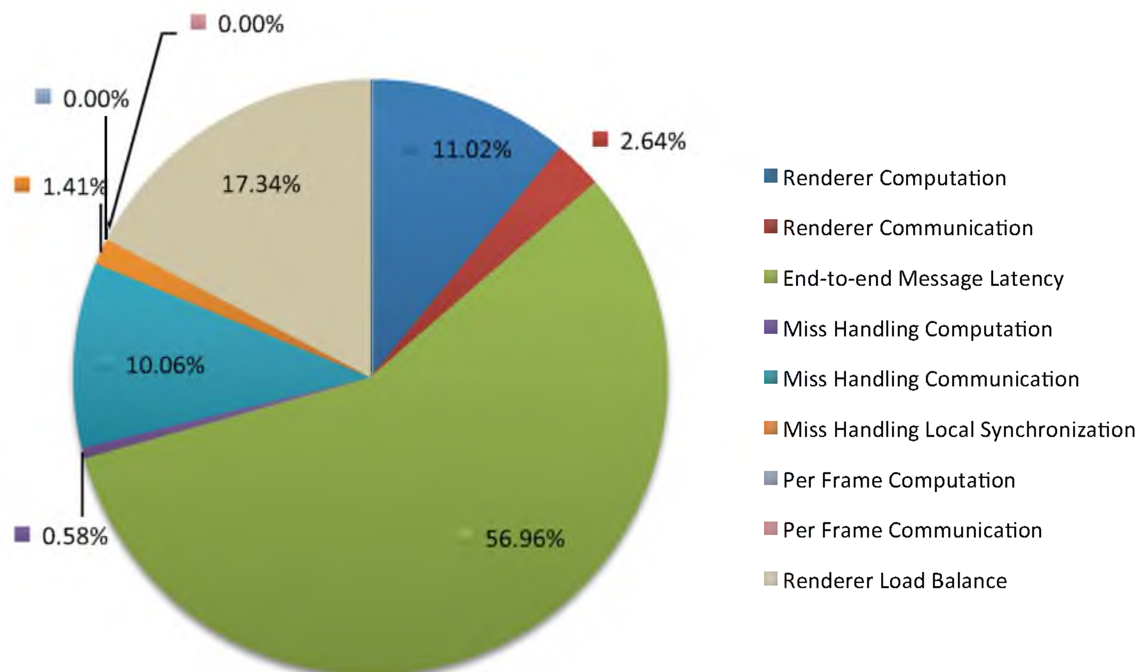

Figure 37: Combined breakdown summary

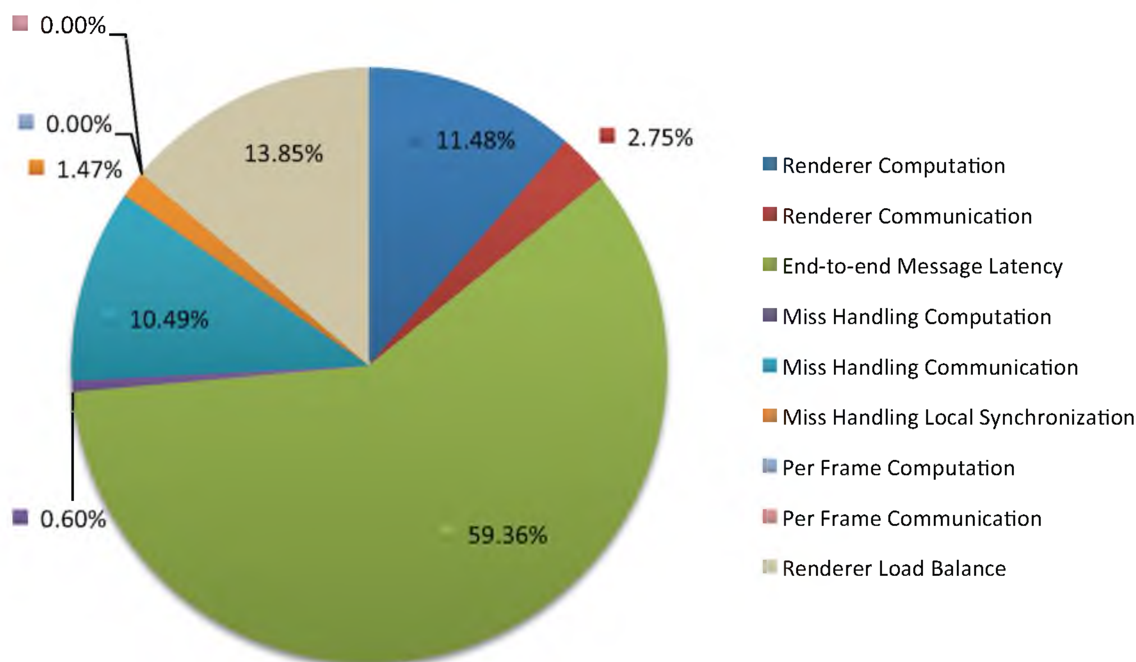

Figure 38: Corrected combined breakdown summary

Table 9: Summary of key `communicator` thread characteristics

Characteristic	Value
Average computation time per cache miss	3.74 μs
Average computation time per task assignment	7.35 μs
Average number of task assignments per node per frame	128
Average computation time per frame per node	57.7 μs
Cache miss response size	3472 bytes
Assignment size	12 bytes
Per Frame Data Size	172 bytes

production machine where other jobs were running. While the nodes in the run were dedicated to the particular run, the network resources were shared. As the effective network bandwidth decreases the overall communications time increases.

The load imbalance also scales super-linearly from 32 to 40 nodes but then becomes fairly random thereafter. This is as expected. As the overall communication system performance is reduced, all nodes are effected equally. Due to the load balancing nature of the

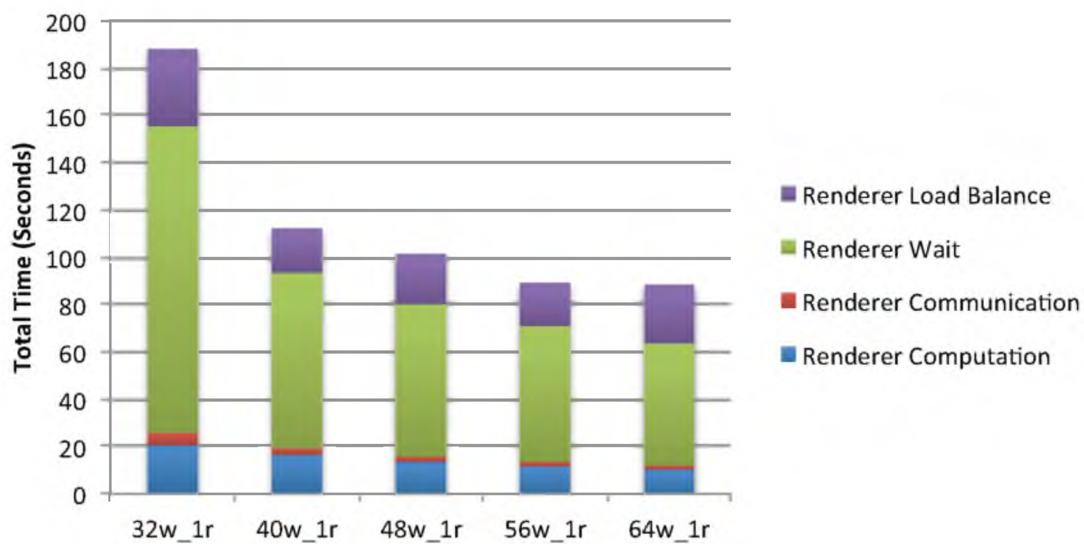

Figure 39: Total render time versus number of nodes

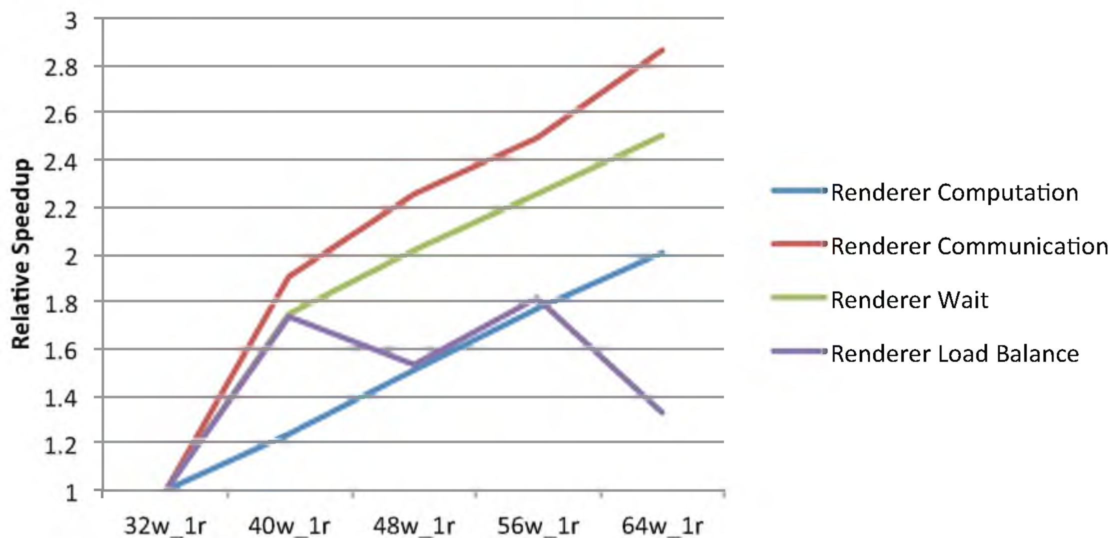

Figure 40: Relative speedupversus number of nodes

work assignments in DIRT, the load imbalance is roughly proportional to the time it takes to complete an assignment. As the communication time increases, the time to render an assignment increases by the same amount. Since the communication time on this cluster dominates the total render time, this results in an increase of the assignment time that is nearly proportional to the increase in communication time.

As can be seen in Figures 41 and 42, as the number of threads per node increases, the performance overall scales negatively. The primary reason for this is that there are only two processor cores per node on this cluster: one for the `communicator` thread and one for the `Renderer` thread. As more threads are added, context switch overhead degrades local compute performance. Furthermore, if the `communicator` thread is not running when a message arrives, that adds to the perceived message latency.

Figures 43 and 44 show that the same basic trends hold for all of these runs. It is clear from the data that for the 56w_2r, 64w_2r, 56w_3r, 64w_3r, 56w_4r, and 64w_4r runs, there was competition for network resources at the time of these runs. It also appears there as some competition during the 23w_1r, 32w_3r, and 32w_4r runs.

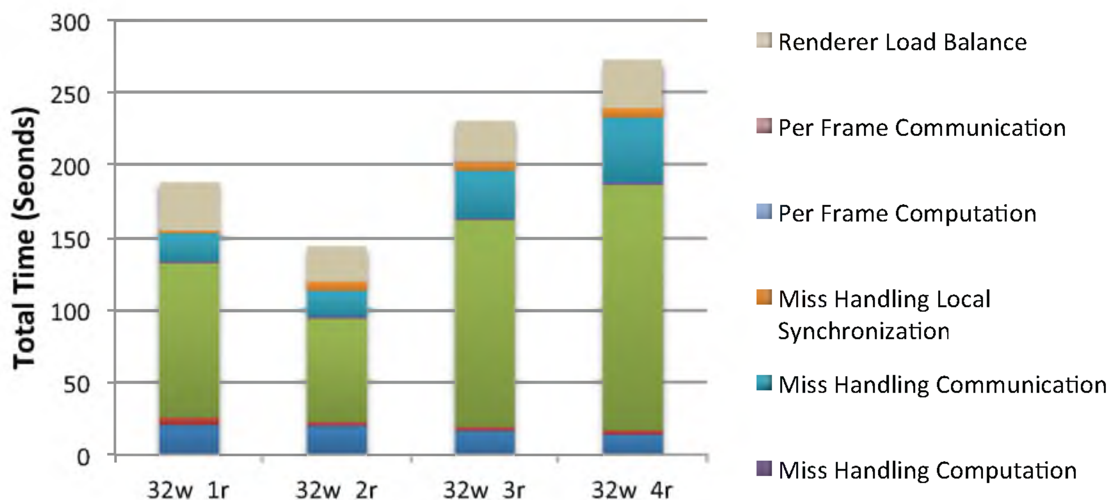

Figure 41: Total render time versus number of `Renderer` threads

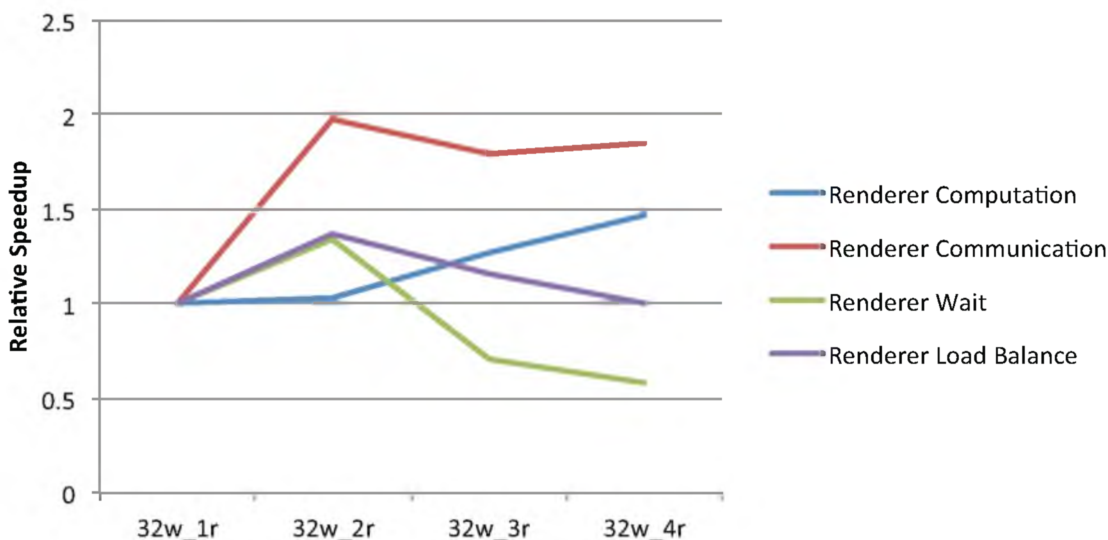

Figure 42: Relative speedup versus number of `Renderer` threads

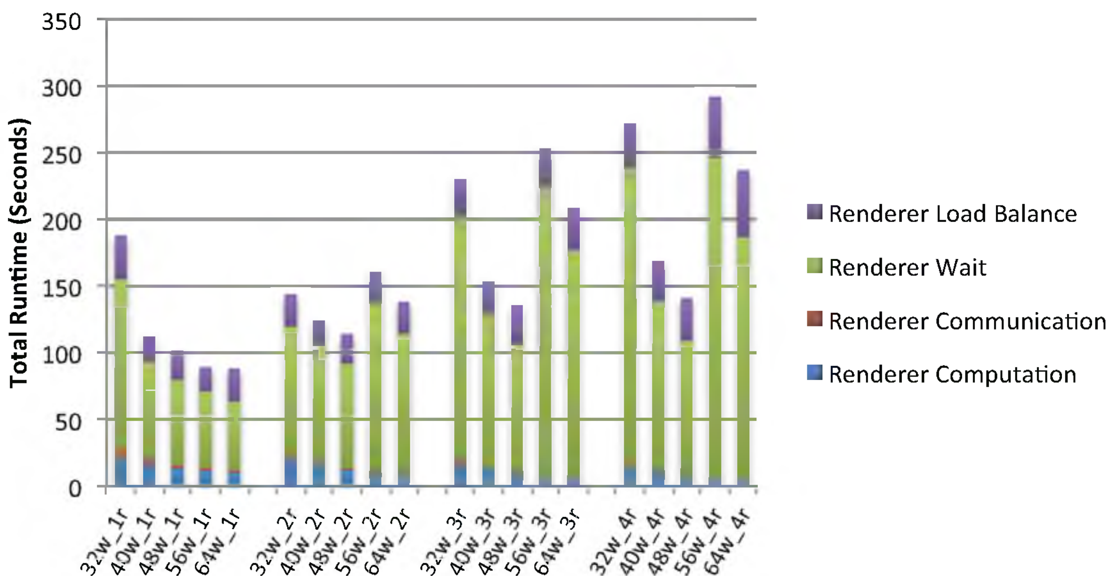

Figure 43: Total time summary versus node count for all runs

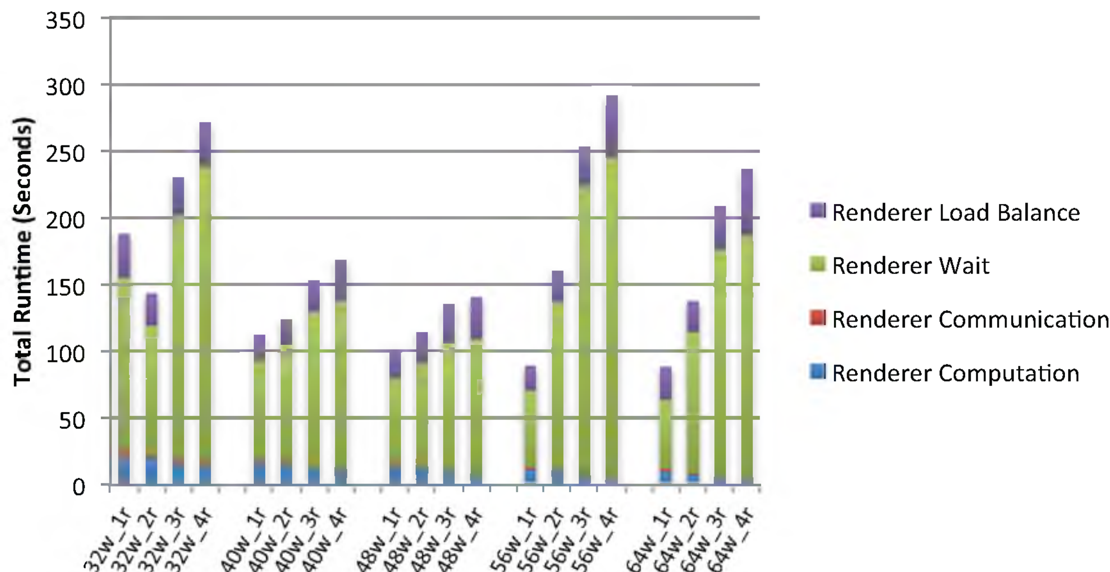

Figure 44: Total time summary versus thread count for all runs

Analogous to Figure 37, Figures 45 and 46 show the combined time spent rendering organized by node count and by thread count, respectively. Again, it can be seen that the total runtime is dominated by end-to-end message latency both in the OS and on the actual network.

6.5 Cache Miss Model

In addition to looking at the performance data across several runs of DIRT, it is important to understand how many software shared memory cache misses occur as a function of the number of nodes and `Renderer` threads. The data for this model was derived from looking at 3 runs for each data point. The cache misses per thread are summed to produce a total count of cache misses across all of the threads on all of the nodes. There are many interactions one might expect as the number of nodes and threads varies. The amount of total work is identical in all cases. As the number of nodes increases, there will be slightly more compulsory misses for common shared data as each node will need a copy of that da-

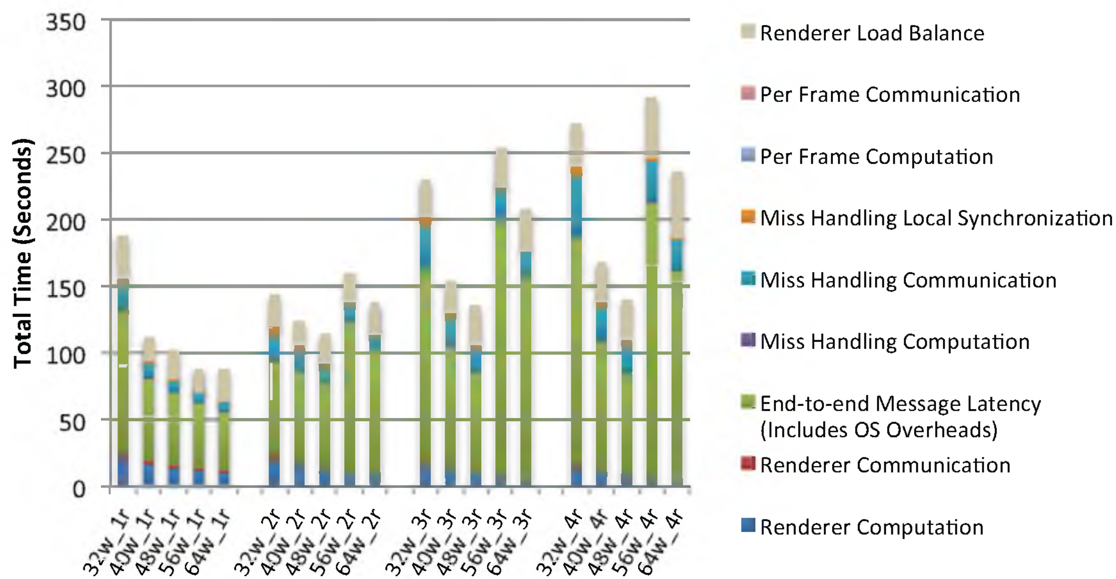

Figure 45: Combined time summary versus node count for all runs

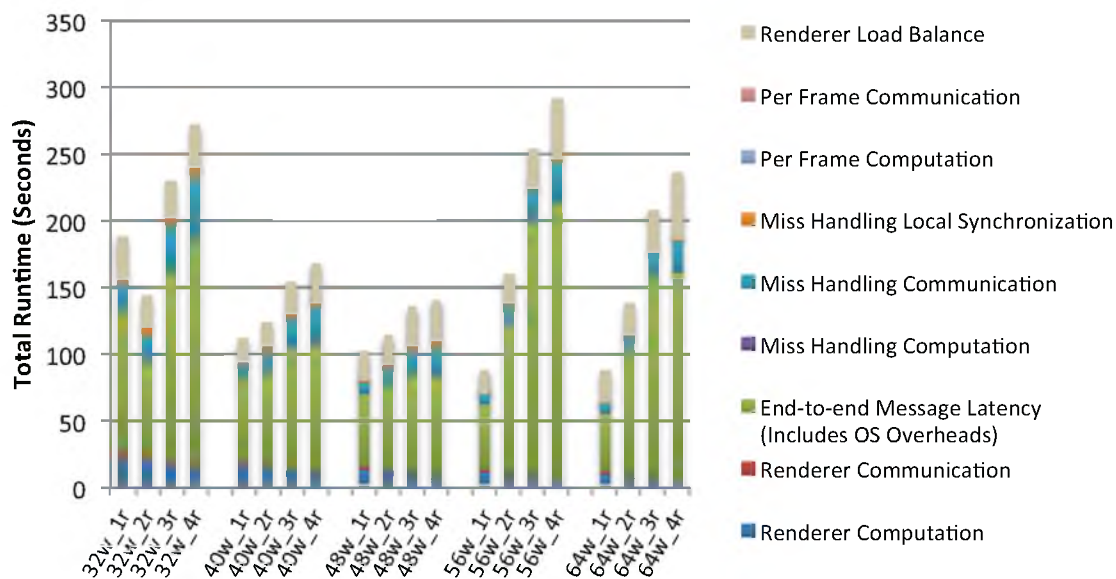

Figure 46: Combined time summary versus thread count for all runs

ta. However, as the cache per node is constant, the total cache capacity increases as the node count increases. In all of the runs, the cache hit rate is fairly high and has a fairly narrow range, from a low of 87.4% to a high of 89.1%. As more threads are added, there is more contention for the local caches. The data obtained from 3 runs of each node and thread count was used to produce a log-linear model that fits the total number of cache misses. The model is,

$$M = 9154620 + 4454.49 \cdot N_n + 823834 \cdot \log(N_r)$$

where M is the total number of misses across all threads and nodes, N_n is the number of worker nodes, and N_r is the number of `Renderer` threads. The fit of that model to the experimental data is shown in Figure 47.

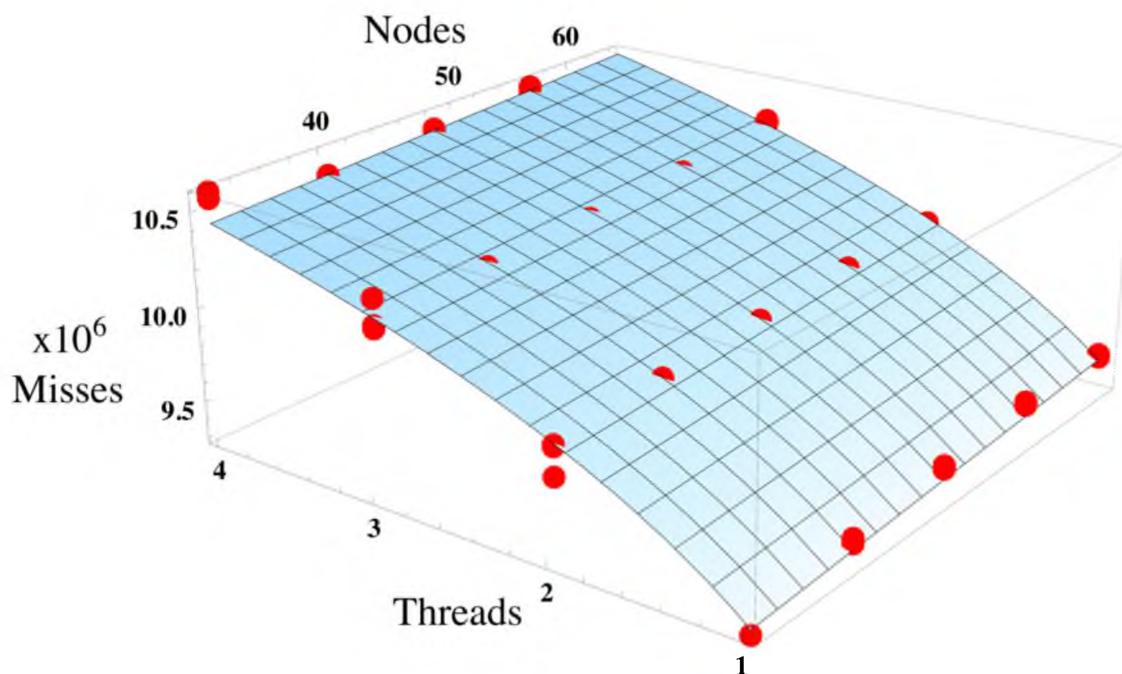

Figure 47: Cache miss model fit

The average number of cache misses per frame per renderer thread, N_M, can be described as,

$$N_M = \frac{9154620 + 4454.49 \cdot N_n + 823834 \cdot \log(N_r)}{N_n \cdot N_r \cdot 110}$$

That per-thread cache miss model is used in the model in Chapter 7.

6.6 Characterization Summary

Using the analysis presented in this section, it is possible to build a model for both the `Renderer` and `communicator` threads of DIRT. The model is based on the view that the `Renderer` thread is a service that waits for an assignment, computes for some amount of time, requests some number of cachelines, waiting for each to be filled, and returns results to the supervisor. The `communicator` thread on each node is a service that listens for messages from other nodes. For each message type, there is an overhead as measured above. The complete mathematical model for DIRT is developed in Chapter 7. The characteristics from the analysis of DIRT that are used in that model are summarized in Table 10.

Table 10: Summary of key DIRT characteristics

Characteristic	Value
Total `Renderer` compute time per frame (all nodes)	6.03 s
Total task assignments per frame (all nodes)	4096
Average `communicator` computation time per cache miss	3.74 µs
Cache miss model	$v_M = \frac{9154620 + 4454N_n + 823834\log(N_r}{N_n \cdot N_r \cdot 110}$
Miss request size	20 bytes
Average pixel assignment send size	796 bytes
Cache miss response size	3472 bytes
Assignment size	12 bytes
Per frame data size	172 bytes

CHAPTER 7

A MATHEMATICAL MODEL OF DIRT

Running a large demanding application such as DIRT is beyond the capabilities of the simulator described in Chapter 4. To predict the performance of DIRT on the ULN architecture, a model has been generated for the worker nodes based on the analysis of the code and the characterization of the 32w_1r run from Chapter 6. That model is based on a simplified compute and communicate view of the operation of the worker node.

7.1 `Renderer` Thread Model

The renderer threads operate as follows:

- For each frame:
  - For each assignment:
    - Wait for the assignment
    - For each pixel in the assignment
      - Compute the ray intersections
      - For each miss:
        - Send a message to a remote node
        - Sleep
        - Wait for wake-up from `communicator` thread

- Wait for frame load imbalance

The total time for the run can be described as:

$$T = n_F(n_A(t_A + n_P(t_P + n_M t_M)) + t_L)$$

where T is the total time, n_F is the number of frames rendered, n_A is the average number of assignments per `Renderer` thread per frame, t_A is the average assignment overhead, n_P is the average number of pixels per assignment, t_P is the average time spent rendering per pixel, n_M is the average number of misses per pixel, t_M is the time spent waiting for the miss to be serviced, and t_L is the expected load-balance wait time per frame.

This can be expanded to:

$$T = n_F n_A t_A + n_F n_A n_P t_P + n_F n_A n_P n_M t_M + n_F t_L$$

As DIRT is intended to be an interactive raytracing tool, it is more meaningful to reason about the effective frame-rate or the time per frame than the total runtime for an arbitrary number of frames. The time per frame, T_F, is,

$$T_F = \frac{T}{n_F} = n_A t_A + n_A n_P t_P + n_A n_P n_M t_M + t_L$$

The $n_A t_A$ component corresponds to the total time getting new assignments, excluding load imbalance at the end of the frame. As the assignments are handed out preemptively, there is no wait time associated with getting the next assignment, only computational overhead. The $n_A n_P t_P$ term corresponds to the total time per frame actually rendering pixels.

As the total number of total assignments per frame in DIRT is currently fixed at 4,096 assignments, n_A is just 4,096 divided by the total number of `Renderer` threads. Furthermore, the total amount of render work is independent of the number of nodes and

threads shown in Chapter 6. Thus it makes sense to combine these terms into the total time spent on rendering tasks. If we let T_R represent the total computation time per `Renderer` thread per frame then,

$$T_R = n_A t_A + n_A n_P t_P$$

The $n_A n_P n_M t_M$ term corresponds to the total time lost to cache misses per frame. That is, it is the average number of cache misses per `Renderer` thread per frame times the expected miss time. If we let N_M represent the total number of misses per `Renderer` thread per frame, and t_R represent the average time computing between misses, then,

$$N_M = n_A n_P n_M$$
$$N_M t_M = n_A n_P n_M t_M$$
$$t_R = \frac{n_A t_A + n_A n_P t_P}{n_A n_P n_M} = \frac{T_R}{N_M}$$

Rewriting T_F in these terms gives us,

$$T_F = T_R + N_M t_M + t_L = N_M(t_R + t_M) + t_L$$

This is the model used as the time per frame. The number of misses per thread per frame comes from the model presented in Chapter 6. The time spent rendering per miss, t_R, is computed as follows. The total time rendering per frame, with communication overheads removed is 6.027299 seconds. This is divided by the P4 versus i7 speedup, 5.53, giving an expected computational time per frame of 1.09 seconds. Dividing this by N_M and by the total number of threads gives the expected rendering time per cache-miss if the thread were running alone on the SMT processor. The miss request send overhead of 26 ns is added to this, giving the raw `Renderer` CPU time between cache misses, T_r. Finally, the expected time of running this work on the shared SMT processor, t_R, is computed by dividing T_r by the expected speedup from multiple threads running, S_r. Thus,

$$T_r = \frac{6.03s}{5.53N_M N_n N_r} + 26ns$$

$$t_R = \frac{T_r}{S_r}$$

The time to service a miss, t_M, is the sum of the round-trip network latency to the home node, $2l$, the serialization delay of the request, b_Q, the serialization delay of the response, b_S, the expected queuing delay of both the request and the response, q, and the expected `communicator` handling time for the request and response, t_H.

$$t_M = 2l + b_Q + b_S + q + t_H$$

where the t_H term is derived from the time expected to handle a single cache miss when running alone on the SMT processor, T_h, and the expected speedup of the handler thread, S_h as,

$$t_H = \frac{T_h}{S_h}$$

The one-way latency, l, is the sum of the wire latency of 300 ns, plus the send side latency of 45 ns, the receive side latency of 103 ns, and the wake-up latency of 5 ns. The serialization latencies are a function of bandwidth. The request size from Chapter 6 is 58 bytes and the response size is 3530 bytes across 3 packets. Thus,

$$l = 300ns + 45ns + 103ns + 5ns = 453ns$$
$$r_Q = 58ns$$
$$r_S = 3530ns$$

T_h is discussed in Section 7.2 and T_r and T_h are used in the model for computing S_r and S_h in Section 7.4. The queuing term, q, and the handling time are combined in the queuing model in Section 7.3.

7.2 Communicator Thread Model

The `communicator` threads operate as follows:

- Wait for incoming message
- Receive message
- Switch on message type:
  - Dataserver request
    - Read dataserver header
    - Switch on request type:
      - Cache miss request
        - Look-up & read local scene data
        - Send response
      - Cache miss response:
        - Receive data
        - Fill cache-line
        - Wake `Renderer` thread
  - Task assignment response
    - Place task in queue

In the ULN architecture, receives are handled by the hardware and placed directly in the user address space. The `communicator` main loop would receive the entire message and pass a pointer to lower-levels of the call-tree for protocol specific packet data. Thus, for each of flows, we can simplify the view such that there is one receive and associated processing.

For the model in Section 7.1, we just need the average service time for a single cache miss, which is the sum of the time spent handling the miss request and the miss response. The total request and response processing time is computed as 3.74 μs in Chapter 6. This is divided by the P4 versus i7 speedup, 5.53, giving an expected computational time per miss of 677 ns. To that we add the communications overheads from Chapter 4. On the request side there is a receive overhead of 32 ns to receive the request and a send overhead of 26 ns to send the response. On the response side, there is a receive overhead to receive the response or 32 ns, and a thread wake overhead of 2 ns to wake the `Renderer` thread. Thus,

$$T_h = \frac{3740ns}{5.53} + 32ns + 26ns + 32ns + 2ns = 769ns$$

7.3 Expected Message Queueing Latency

As worker nodes render the scene, they must request scene data from other nodes. They do this by sending a request message to the home node for the data and then wait for a response. A simple model just modeling the average latency of each one-way message along with a simple average overhead model is sufficient to capture most of the message behavior. However, at each home node, there may be multiple requests that arrive in near succession to each other, causing some queueing delay in the system. A queueing model is applied to account for this behavior.

A model for a bounded single queue is used as a model for the message handler routine used at each node. The common model, known as an M/M/1/B model [56] assumes that requests arrive at the handler with some average rate, λ, and are handled at some average rate, μ, each with an exponential distribution. A bounded queue is modeled here due to

the fact that there are a limited number of workers. For a low traffic intensity, ρ, where $\rho=\lambda/\mu$ is much less than 1, and where the number of potential requests is relatively high, an unbounded queue is also a very good approximation of a bounded queue.

The model for an M/M/1/B queue is as follows. The expected response time is:

$$E[r] = \frac{E[n]}{\lambda(1-p_B)}$$

where, $E[n]$, the expected number of jobs in the system is,

$$E[n] = \frac{\rho}{(1-\rho)} - \frac{(B+1)\rho^{(B+1)}}{1-\rho^{(B+1)}}$$

where, p_B, the probability of the maximum, B, jobs in the system is,

$$p_B = \frac{1-\rho}{1-\rho^{B+1}}\rho^B$$

Thus, the expected response time can be modeled as,

$$E[r] = \frac{\frac{\rho}{1-\rho} - \frac{(B+1)\rho^{B+1}}{1-\rho^{B+1}}}{\lambda\left(1 - \frac{1-\rho}{1-\rho^{B+1}}\rho^B\right)}$$

The value of B is the maximum number of outstanding messages that can be queued at a node. In this system, there can be one request message from each of the workers not local to the node, plus one response for each worker on the local node, therefore B is equal to the total number of workers in the run. As can be seen, as ρ^{B+1} approaches zero, this approaches the equation for an infinite length M/M/1 queue model [56], where,

$$E[r] = \frac{\rho}{\lambda(1-\rho)} = \frac{1}{\mu(1-\rho)} = \frac{1}{\mu-\lambda}$$

This equation is useful for a quick and simple understanding of the queuing delay. It can only overestimate the latency of handling a message, and does so by a surprisingly small amount. In our case, ρ is approximately 0.11 or lower. The interesting values of B are

those greater than or equal to 32. This means the value of ρ^{B+1} is on the order of 10^{-32} or less. Therefore the error, or over estimation from using the simplified model in this work is trivial. The value of having a simplified equation is well worth the negligible error.

E[r] is the sum of the queuing delay and the handling delay. The following equations relate μ, λ, and *E[r]* to the terms in Section 7.1.

$$E[r] = q + t_H$$

$$\lambda = \frac{1}{t_R + t_M}$$

$$\mu = max(\frac{1}{t_H}, \frac{1}{b})$$

7.4 Multiple Thread Performance Model

A necessary component of the model of an SMT application is a model for how the threads interfere and subsequently reduce the speed of each other. From a single thread perspective, a `Renderer` thread computes for some time and then blocks for some amount of time waiting for a communication response. On a single threaded processor, the ratio of the compute time to communication time can be combined with measured values for how a SMT processor performs with multiple threads running to understand the performance of the rendering step of DIRT as a function of those parameters. Having a mathematical model for this as opposed to just measuring it is necessary as the ratio of compute to communicate time is itself a function of the latency and overhead of the communication system.

On a SMT processor, capable of running N threads simultaneously, for a specific workload, there is a per thread speedup, S_n, when n of the N threads are active relative to when only one thread is active. If each thread, i, computes for some time T_i, as measured when only thread i is active, and then stalls for some time T'_i, an equation describing the

histogram of how often the processor is running 0 to N threads can be formulated using a rate-based Markov model. Where $T'_0=T'_1=T'_N=T'$ and $T_0=T_1=T_N=T$, the model is a simple birth-death process as shown in Figure 48.

In that model, p_i is the probability of being in a state i where i threads are running concurrently. In the steady state, the probability of a state times the sum of rates of all edges flowing out is equal to the sum of the product of the probability of the feeder states times the edge rate flowing the state. Thus,

$$\frac{N-i}{T'}p_i = \frac{(i+1)S_{i+1}}{T}p_{i+1}$$

or

$$p_{i+1} = \frac{\frac{N-i}{T'}}{\frac{(i+1)S_{i+1}}{T}}p_i = \frac{(N-i)T}{(i+1)T'S_{i+1}}p_i$$

Furthermore, the sum of the probabilities must be 1.

$$1 = \sum_{i=0}^{N} p_i$$

If we define S_0=1 for convenience, the solution to that set of equations yields,

$$p_i = \frac{\binom{N}{i}\frac{T^i}{T'^i\prod_{j=0}^{i} s_j}}{\sum_{k=0}^{N}\binom{N}{k}\frac{T^k}{T'^k\prod_{j=0}^{i} s_k}}$$

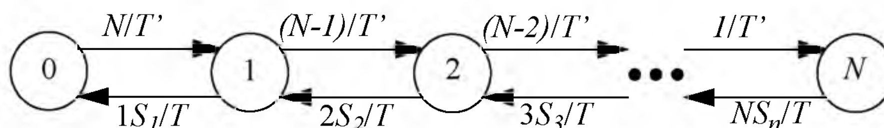

Figure 48: Markov model of isomorphic threads on an SMT processor

The combinatorics in the summation account for all possible pairings of threads that can run simultaneously.

On a DIRT worker node, there are N-1 or N_r `Renderer` threads and 1 `communicator` thread. The `Renderer` threads all have the same profiles. However, the `communicator` thread has a distinct profile. Thus the model must account for the difference. It is not necessary to enumerate all of the possible of threads combinations, as all `Renderer` threads are equivalent. However, it important to distinguish when the `communicator` thread is and is not part of the mix. In this case, we will use T_r as the expected time it takes to render a chunk before communication and T_c as the expected time for the communication round-trip to characterize the `Renderer` threads. Likewise, we will use T_h as the expected time for the `communicator` thread to handle a request and T_m as the expected time until the next message arrives. The states named 0, 1, ..., i, ..., N_r are the states where there the `communicator` thread is not running and i `Renderer` threads are running concurrently. The states named 0c, 1c, ..., ic, ..., $N_r c$ are the states where the `communicator` thread is running concurrent with i `Renderer` threads. The Markov model for this case is shown in Figure 49.

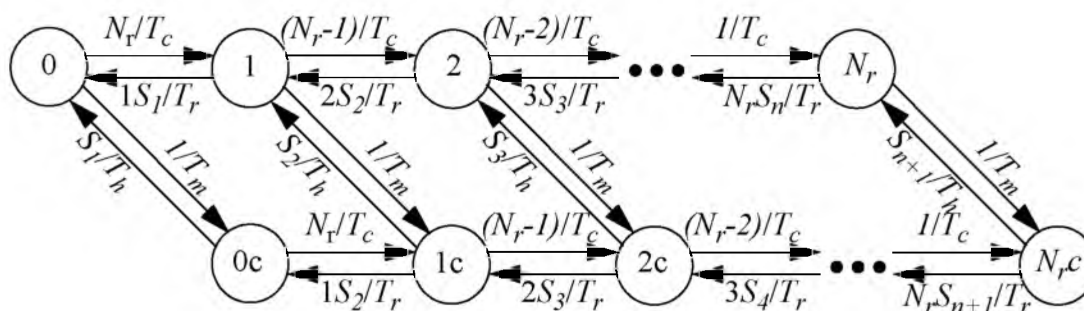

Figure 49: Markov model of DIRT threads on an SMT processor

The solution to this yields,

$$p_i = \frac{\binom{N_r}{i}\dfrac{T_r^i}{T_c^i\prod_{j=0}^{i} s_j}}{\sum_{k=0}^{N_r}\left[\binom{N_r}{k}\dfrac{T_r^k}{T_c^k\prod_{j=0}^{k} s_j}\right] + \sum_{k=0}^{N_r}\left[\binom{N_r}{k}\dfrac{T_r^k T_h}{T_c^k T_m\prod_{j=0}^{k} s_{j+1}}\right]}$$

$$p_{ic} = \frac{\binom{N_r}{i}\dfrac{T_r^i T_h}{T_c^i T_m\prod_{j=0}^{i} s_{j+1}}}{\sum_{k=0}^{N_r}\left[\binom{N_r}{k}\dfrac{T_r^k}{T_c^k\prod_{j=0}^{k} s_j}\right] + \sum_{k=0}^{N_r}\left[\binom{N_r}{k}\dfrac{T_r^k T_h}{T_c^k T_m\prod_{j=0}^{k} s_{j+1}}\right]}$$

These equations were checked by Monte Carlo simulation where synthetic "threads" requested the "processor" for a random amount of time centered around T, T_r, or T_h as appropriate and then stalled for a random amount of time centered around T', T_c, or T_m as appropriate. After running for 1,000,000 cycles, the simulation results agree within 1% of the calculated results for a range of parameters.

This equation can be used to compute the expected speedup of the threads running on the processor. The expected speedup is the sum of the speedup while running i threads, S_i, times the probability of i threads running, p_i. For the renderer threads, the expected speedup $E[S_r]$ is,

$$E[S_r] = \frac{\sum_{i=0}^{N_r}\frac{i}{N_r}S_i p_i + \sum_{i=0}^{N_r}\frac{i}{N_r}S_{i+1}p_{ic}}{\sum_{i=0}^{N_r}\frac{i}{N_r}p_i + \sum_{i=0}^{N_r}\frac{i}{N_r}p_{ic}}$$

The expected speedup of the **communicator** thread handling a message, $E[S_h]$, abbreviated S_h, is,

$$S_h = E[S_h] = \frac{\sum_{i=0}^{N_r} S_{i+1} p_{ic}}{\sum_{i=0}^{N_r} p_{ic}}$$

The expected speedup depends on T_m, which depends on the queuing delay from Section 7.3. However, μ, the average rate of handling a request depends on the expected speed of the handler, and hence on S_h. The arrival rate for messages, λ, depends on the total time it takes to run a render and handle cycle. Thus these equations must be solved simultaneously.

T_r and T_h are parameters taken from the model in Sections 7.1 and 7.2 combined with the data in Chapter 6. Specifically, T_r is the expected time rendering before communicating with only a single thread running. T_h is the expected time handling a request and a response from a cache miss communication.

T_c is the expected time waiting for a remote cache miss to be handled and returned. This is equal to the round-trip wire latency of the message traversal, l, the round-trip wire latency including message serialization, plus the expected message handler response time. The expected message handler response time of the **communicator** threads comes from the queueing model explained in section Section 7.3.

$$T_c = l + \frac{1}{\mu - \lambda}$$

$$T_c = l + \frac{1}{max(\frac{S_h}{T_h}, \frac{1}{b}) - \frac{N_r}{T_c + \frac{T_r}{S_r}}}$$

The solution to this equation is,

$$T_c = \frac{1}{2}\left[max(\frac{T_h}{S_h}, b)(N_r+1) + l - \frac{T_r}{S_r} + \sqrt{\left(\frac{T_r}{S_r} - max(\frac{T_h}{S_h}, b)(N_r+1) - l\right)^2 - 4\left(max(\frac{T_h}{S_h}, b)lN_r - max(\frac{T_h}{S_h}, b)\frac{T_r}{S_r} - l\frac{T_r}{S_r}\right)}\right]$$

The total time for a renderer thread to do a computation and communication pass is $T_c + T_r/S_r$. Each renderer thread is issuing a new communication request every $T_c + T_r/S_r$ seconds. Each of the renderer threads spreads its requests approximately equally amongst the `communicator` threads. The ratio of renderer threads is N_r.

At the `communicator`, the expected time until a new message arrives, *Tm*, is,

$$T_m = \frac{T_c + \frac{T_r}{S_r}}{N_r} - \frac{T_h}{S_h}$$

7.5 Load Imbalance

The average load imbalance per processor for each frame can be modeled as the delta of the expected maximum render time and the average render time per frame. The average or mean time spent rendering a frame is straightforward to compute. Extreme value theory [90,105,26] is applied to understand and model the expected maximum render time per frame across *n* worker threads.

The time it takes a single processor to render a single frame is equal to the time the processor takes to render the sum of its assigned tasks. Tasks are assigned to a processor essentially at random and are therefore independent of each other. Thus the time it takes a processor to render a frame can be described statistically as a random variable with an associated probability density function (PDF). While the time distribution of the individual tasks is not Gaussian, the sum of the many tasks assigned to each processor is approximately Gaussian as would be expected under the central limit theorem [90,105].

The set of times it takes the n processors to complete the assigned tasks is viewed as n samples from the normal distribution with a PDF of $f(x)$.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-u)^2}{2\sigma^2}}$$

where the mean time to complete a set of tasks is μ and the standard deviation is σ. The cumulative distribution function (CDF), or the probability that a given sample is less than or equal to x, of this PDF is the integral from negative infinity to x of the PDF.

$$F(x) = \int_{-\infty}^{x} f(y)dy = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-u)^2}{2\sigma^2}} dy$$

Given a set of n samples, the probability that all of these samples are less than some value X_{max} is the product of the probability that each of these samples is less than X_{max}.

$$G_n(x) = \prod_{i=1}^{n} F(x) = F(x)^n = \left[\int_{-\infty}^{x} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-u)^2}{2\sigma^2}} dy\right]^n$$

By definition, the PDF of the maximum value is the derivative of the CDF.

$$g_n(x) = \frac{dG(x)}{dx} = nf(x)F(x)^{n-1}$$

The expected extreme value as a function of the number of experiments can then be computed as,

$$E[g_n(x)] = \int_{-\infty}^{\infty} x g_n(x) dx$$

Which expands to

$$E[g_n(x)] = \int_{-\infty}^{\infty} nxf(x)F(x)^{n-1}dx$$

$$E[g_n(x)] = \int_{-\infty}^{\infty} nx\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-u)^2}{2\sigma^2}}\left[\int_{-\infty}^{x}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(y-u)^2}{2\sigma^2}}dy\right]^{n-1}dx$$

This can be simplified as follows by letting $v = \frac{y-\mu}{\sqrt{2\sigma^2}}$ then,

$$y = \sqrt{2\sigma^2}v + \mu$$

$$dy = \sqrt{2\sigma^2}dv$$

$$E[g_n(x)] = \int_{-\infty}^{\infty} nx\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-u)^2}{2\sigma^2}}\left[\int_{-\infty}^{\frac{x-\mu}{\sqrt{2\sigma^2}}}\frac{\sqrt{2\sigma^2}}{\sqrt{2\pi\sigma^2}}e^{-v^2}dv\right]^{n-1}dx$$

If we let $w = \frac{x-\mu}{\sqrt{2\sigma^2}}$ then,

$$x = \sqrt{2\sigma^2}w + \mu$$

$$dx = \sqrt{2\sigma^2}dw$$

$$E[g_n(x)] = \int_{-\infty}^{\infty} n\frac{\sqrt{2\sigma^2}w+\mu}{\sqrt{2\pi\sigma^2}}e^{-w^2}\left[\int_{-\infty}^{w}\frac{\sqrt{2\sigma^2}}{\sqrt{2\pi\sigma^2}}e^{-v^2}dv\right]^{n-1}\sqrt{2\sigma^2}dw$$

This simplifies and expands as follows,

$$E[g_n(x)] = \int_{-\infty}^{\infty} n\frac{\sqrt{2\sigma^2}w+\mu}{\sqrt{\pi}}e^{-w^2}\left[\int_{-\infty}^{w}\frac{1}{\sqrt{\pi}}e^{-v^2}dv\right]^{n-1}dw$$

$$E[g_n(x)] = \int_{-\infty}^{\infty} n\frac{\sqrt{2\sigma^2}w}{\sqrt{\pi}}e^{-w^2}\left[\int_{-\infty}^{w}\frac{1}{\sqrt{\pi}}e^{-v^2}dv\right]^{n-1}dw + \mu\int_{-\infty}^{\infty} n\frac{1}{\sqrt{\pi}}e^{-w^2}\left[\int_{-\infty}^{w}\frac{1}{\sqrt{\pi}}e^{-v^2}dv\right]^{n-1}dw$$

$$E[g_n(x)] = \int_{-\infty}^{\infty} n\frac{\sqrt{2\sigma^2}w}{\sqrt{\pi}}e^{-w^2}\left[\frac{1}{2}(erf(w)+1)\right]^{n-1}dw + \mu\int_{-\infty}^{\infty} n\frac{1}{\sqrt{\pi}}e^{-w^2}\left[\frac{1}{2}(erf(w)+1)\right]^{n-1}dw$$

If we let

$$I_n = \int_{-\infty}^{\infty} n\frac{w\sqrt{2}}{\sqrt{\pi}}e^{-w^2}\left[\frac{1}{2}(erf(w)+1)\right]^{n-1}dw$$

and let

$$J_n = \int_{-\infty}^{\infty} n \frac{1}{\sqrt{\pi}} e^{-w^2} \left[\frac{1}{2}(erf(w) + 1)\right]^{n-1} dw$$

then this can be written as,

$$E[g_n(x)] = \sigma I_n + \mu J_n$$

We can show that J_n is equal to one for all n by letting

$$u = \frac{1}{2}(erf(w) + 1)$$

then,

$$du = \frac{1}{\sqrt{\pi}} e^{-w^2} dw$$

$$J_n = \int_0^1 n u^{n-1} du$$

$$J_n = u^n \Big|_0^1 = 1$$

This is as expected since J_n is equivalent to a PDF where μ=1 and σ=1, which, by definition, integrates to 1 over the entire range.

Using this result, the expected extreme value equation can be further simplified to:

$$E[g_n(x)] = \sigma I_n + \mu = \sigma \int_{-\infty}^{\infty} n \frac{\sqrt{2}w}{\sqrt{\pi}} e^{-w^2} \left[\frac{1}{2}(erf(w) + 1)\right]^{n-1} dw + \mu$$

The only dependence on σ is that the extreme value scales linearly with σ and the only dependence on μ is that the expected value is offset by μ. This shows that the expected maximum value out of n samples from a Gaussian random distribution can be expressed as μ plus a scale, I_n, times σ. By symmetry, the expected minimum value is μ minus σ times

I_n. Thus for the purposes of understanding load imbalance, the expected time between the shortest and longest task from a sample set is $2I_n\sigma$.

A closed form solution for I_n for values of n greater than 5 is not known [136]. Therefore this integral is evaluated numerically using Mathematica for a range of values of n. These calculations were also verified by a simple experiment by producing N=100,000 sets of n values from a normal distribution, by using a Box-Mueller transform [15] on uniform random values generated by `random()`, and comparing the average maximum value across the N experiments with the analytical expected maximum value. For all experiments, the values compared are within 0.27% of the analytical model.

Figure 50 shows the values of I_n that are of the most interest for the experiments in this work. At 32 workers, I_n is about 2.07 and the expected delta between the minimum and maximum render times on a particular frame. Note that the x-axis is logarithmic.

Figure 51 shows how this function continues to grow to much larger values of n. As can be seen, this function grows sublogarithmically. However, it is not asymptotic. As n approaches infinity, I_n also approaches infinity. That is, with an infinite number of samples, the set will include the extremes at plus and minus infinity according to the Gaussian distribution.

For fixed values of σ and μ, load imbalance grows slowly as a function of processor count. How μ and σ change as the number of nodes scales is problem dependent. If μ shrinks and σ remains constant, time spent in load imbalance ($2\sigma I_n/\mu$) grows more quickly. For the DIRT evaluation in this work, the task size is fixed. Thus for a given architecture the ratio of μ to σ is roughly fixed. As communication time shrinks, the mean task time shrinks and the variance also shrinks slightly sublinearly. Thus, the percentage of time

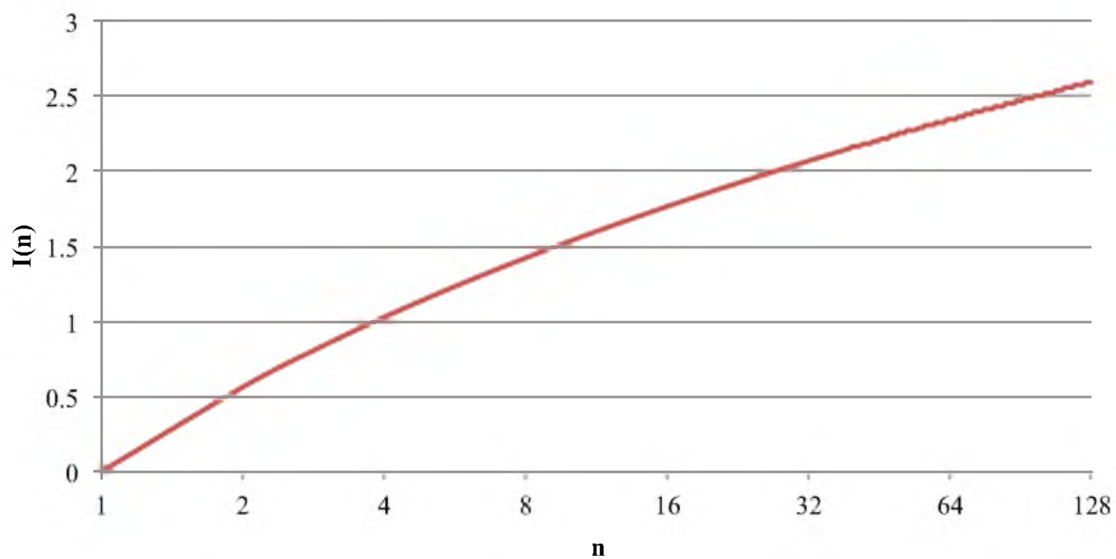

Figure 50: Value of I_n for n=1 to n=128

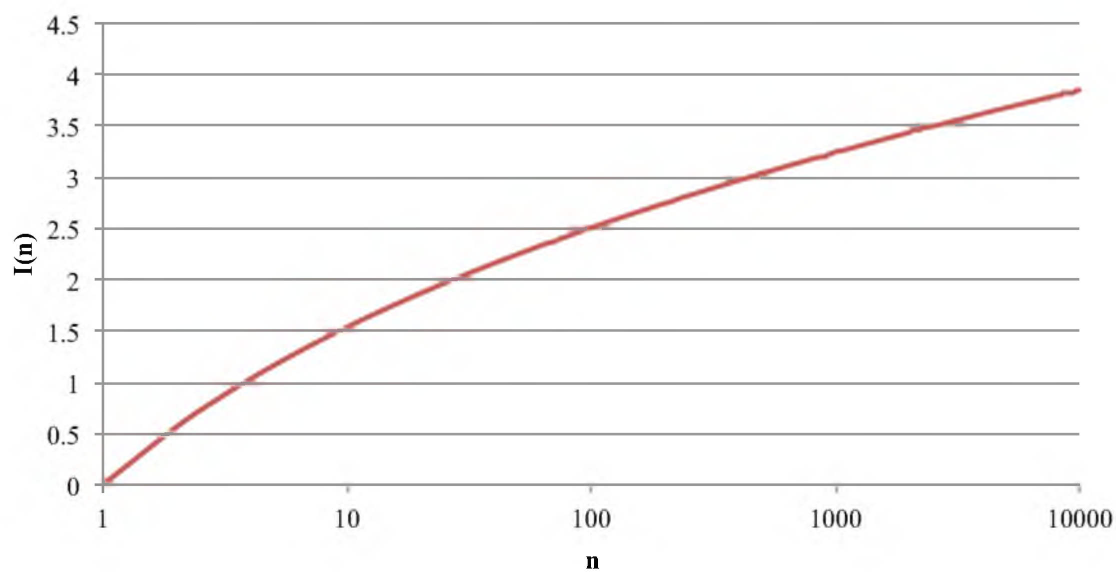

Figure 51: Value of I_n for n=1 to n=10,000

spent in load imbalance tends to be nearly constant, getting slightly worse as the communication overhead approaches zero. This slightly dampens the improvement seen as the communication overhead approaches zero, however, the impact is small for practical values of overhead.

This formula is particularly insightful when used as a ratio. The ratio of I_n at 64 threads to I_n at 32 threads is a factor of 1.13. The ratio of I_n at 512 threads to I_n at 32 threads is a factor of 1.471. This means that the approximate load imbalance will increase from the measured 13% overhead at 32 nodes to 14.7% at 64 threads and 19.1% at 512 threads. This 13% measured baseline is the baseline used in this model.

7.6 Combined Model

The queuing model and the speedup model have an interdependence. More threads executing simultaneously on the SMT processor implies that each of those threads run slower. As the communications thread runs slower, the total time between misses increases, as both T_r and T_h increase, lowering the arrival rate in the queuing model. Since there is an interdependence, the equations are solved iteratively until the error is significantly smaller than 1 ns.

In addition to the parameters already described, the model includes parameters that can be used to add latency and overheads to the system. These parameters include additional T_r and T_h overheads and additional latency in the network. These parameters are used in Chapter 8 to show sensitivities of the architecture to various overheads.

CHAPTER 8

DIRT ON ULN

There are several aspects of the ULN architecture that can be examined in the context of DIRT using the model presented in Chapter 7. These include the impact of the multiple threads per node and the benefits of the various aspects of the architecture. Each of these studies are presented in this chapter. The average render time per thread for the 32w_1r case on the ULN architecture is projected at 56.0 milliseconds versus the 1.69 seconds measured on the cluster in Chapter 6. Some fraction of that 30.2x speedup is due to the projected speedup of 5.53x of a modern core running at 3 GHz versus the older and slower Pentium 4 cores running at 2.4 GHz on the cluster during the compute portions of the code. However, the primary saving is due to significant improvements in the communication architecture.

8.1 SMT Performance

As seen in Section 4.4, the SMT processor modeled in the ULN simulator has an aggregate speedup through at least the 8 threads modeled. In DIRT, the `communicator` thread is shared by all remote `Renderer` threads. As the number of local `Renderer` threads increases, competition for the SMT processor increases and the per-threads speedup decreases. The per-thread speedups of the `Renderer` threads (S_r) and the `communicator` thread (S_h) are shown in Figure 52.

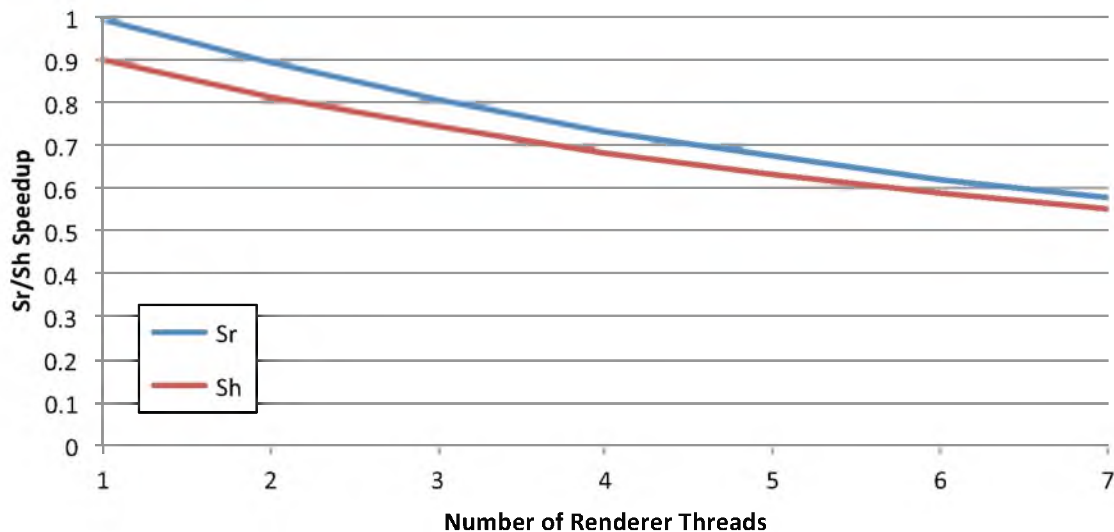

Figure 52: Per-thread speedup as a function of thread count

As the number of remote `Renderer` threads increases, the demand for services from the single `communicator` thread increases and the expected speed of the `communicator` thread decreases. This causes the queuing delay for the `communicator` thread to increase. Furthermore, the contention for the processor resources also causes the `Renderer` threads to operate slower. While this diminishes some of the benefit of the additional threads, Figure 53 shows that there is still an application speedup as the number of `Renderer` threads increases.

This indicates that having more threads per SMT processor in the ULN architecture has a potential benefit. There are second order-effects and physical constraints that may reduce the effectiveness of higher thread counts and place a practical limit on the number of threads that make sense. The actual data patterns and instruction mix in DIRT do not perfectly match the workload investigated in the SMT thread performance model in Chapter 4. This may result in the aggregate throughput to fall-off more sharply and eventually even regress as the thread count increases.

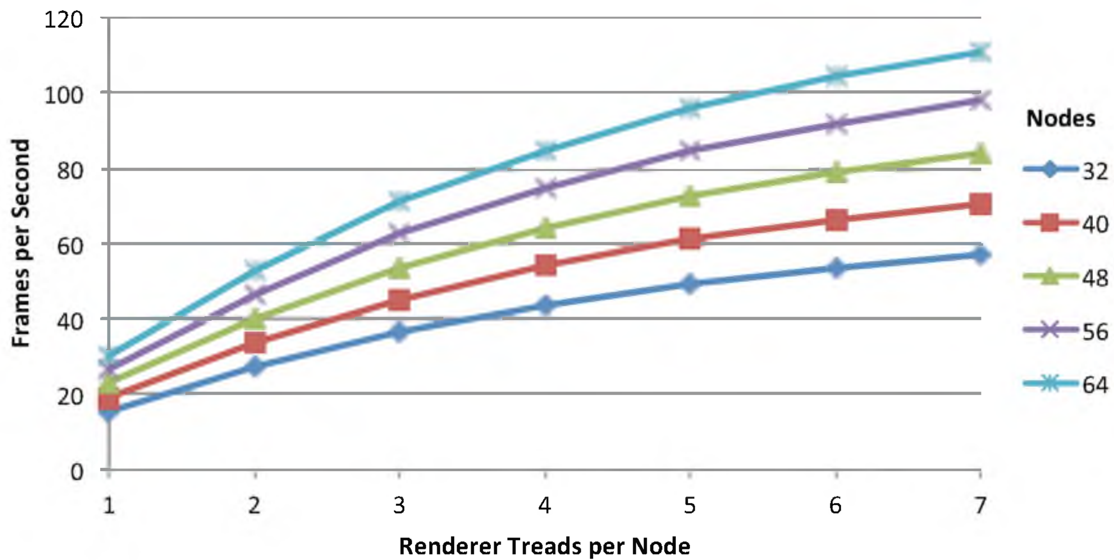

Figure 53: SMT speedup at multiple nodes

As the number of threads increases, the size of the register file and various control structures around the processor also increase approximately linearly. As they increase, the power for these structures also increases proportionally. This results in a lower dynamic power efficiency per operation. As the size of these structures increase, they put pressure on static timing and eventually result in a reduction of the processor's maximum clock frequency. This will further reduce the benefits of increased thread counts. However, the increased number of threads decreases the expected idle time of the processor and increases performance, thus the static power component is amortized more effectively.

Figure 54 shows the probabilities of being in each of the states, *i*, and *ic*, in Markov model from Figure 49 in Section 7.4. The math without the feedback from the queuing and message handling model inSection 7.3 causes the probabilities to weight more heavily toward higher values of *i*. The feedback caused by the cache-miss iteration cycle lengthening as the effective speedups increase lessens the probabilities at the higher values of *i*. The re-

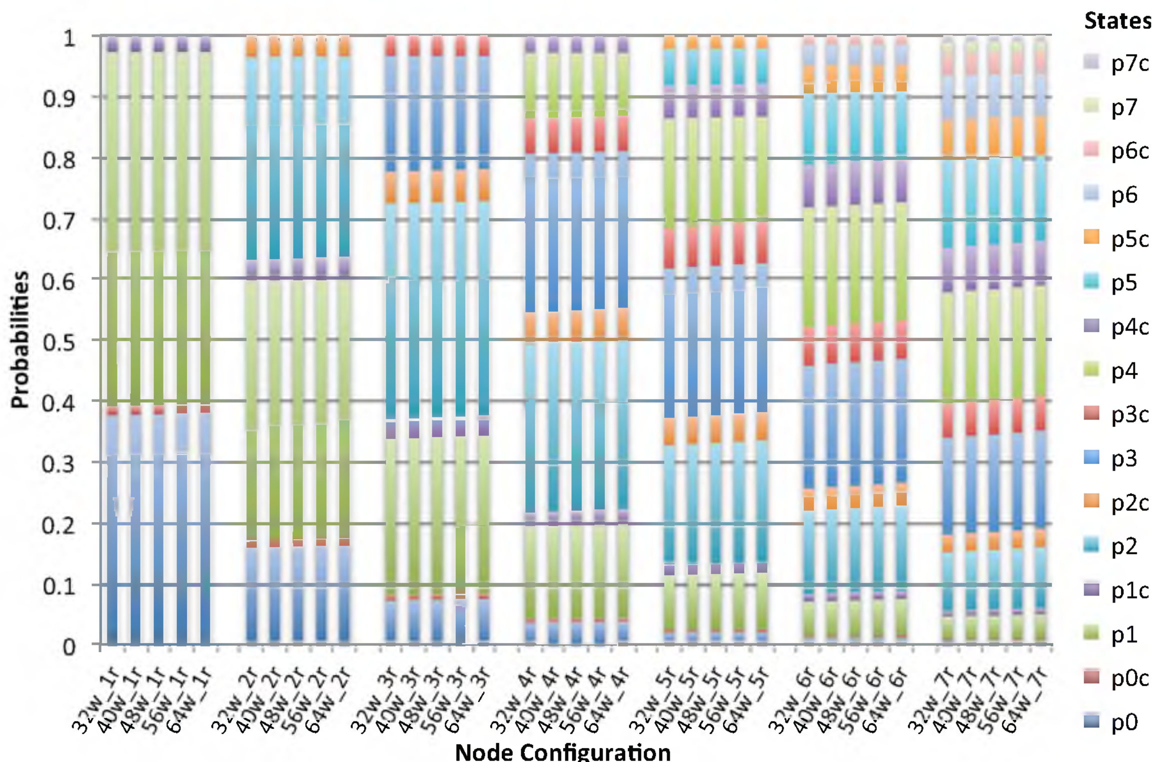

Figure 54: Markov model state probabilities

sult is that there is a greater probability that time will be spent more toward the middle of the thread count range.

By looking at the same data from Figure 53 organized by thread counts, Figure 55 shows that the performance return of adding threads diminishes quickly beyond 3 or 4 `Renderer` threads. Future work should also look at an energy model for the ULN processor to suggest an optimal number of SMT threads from a power perspective.

From these results, a total number of 4 SMT threads appears to strike a reasonable balance between performance and keeping thread counts within reasonable bounds in the ULN architecture. For DIRT, this implies one `communicator` threads and 3 `Renderer` threads. While other thread counts are discussed below, the primary focus will center around this configuration.

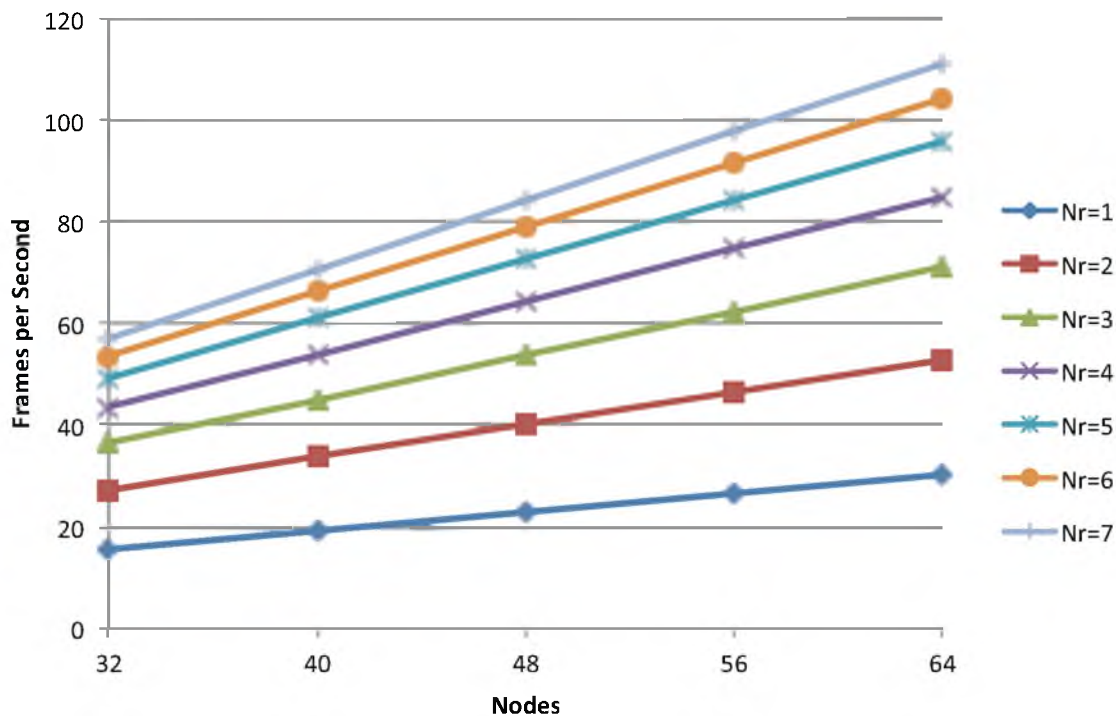

Figure 55: Per-thread speedup as a function of thread count

8.2 Scalability

Figure 55 gives an indication of the scalability of the architecture. The results indicate a scaling efficiency of 97.5% for N_r=1 to 97.4% for N_r=7. The scaling efficiency is dropping very slowly as the number of `Renderer` threads increases. Figure 56 shows the performance as the node count scales from 8 nodes to 512 nodes.

Looking at node counts up to 512 nodes pushes the model to its extreme. The performance the model is based on is for node counts from 32 to 64 nodes. This extends some of the data from those trends a factor of 4 below the bottom-end and a factor of 8 above the top-end. At the top-end one concern is that the total number of threads is nearly equal to the total number of assignments per frame. Thus the load-balancing features of DIRT will not have enough assignments to operate effectively. As the overhead per assignment is small, increasing the number of assignments per frame by decreasing the size of the assignments

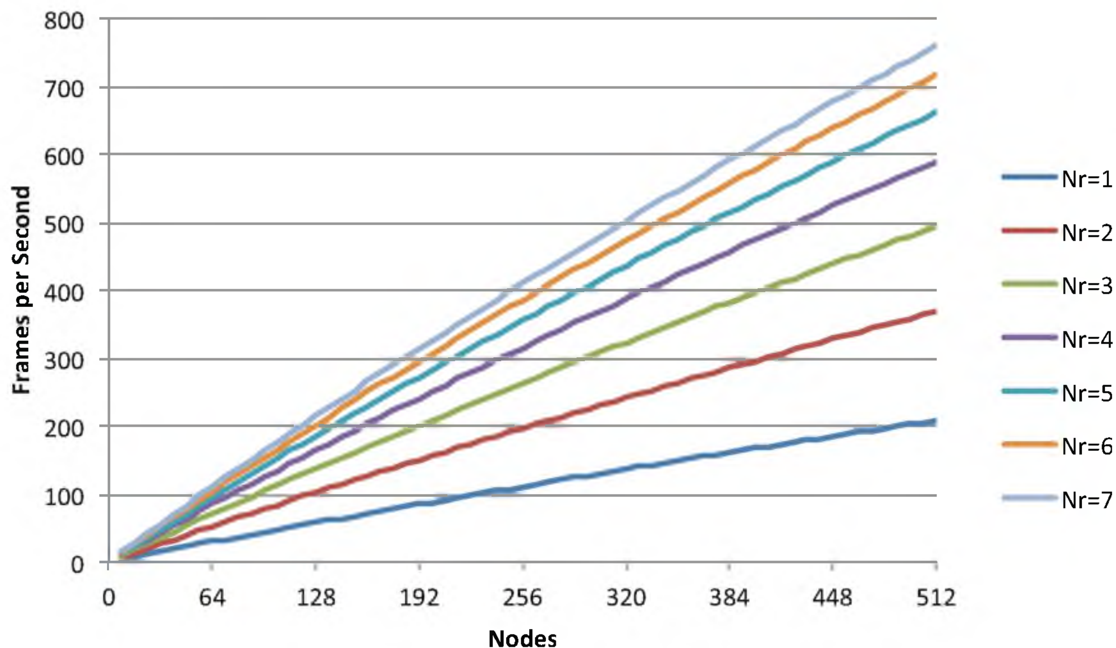

Figure 56: Scaling to 512 threads

would be reasonable. This would result in a slightly higher cache miss rate that should be similar to the increased miss rate observed by increasing the node count. So while the estimates toward the higher node and thread counts are probably an over-estimation of the performance, the architecture clearly supports good scaling for N_r=3 up to at least 4 times the node counts measured.

Figure 57 show the scaling efficiency of the architecture on DIRT. On the left is the speedup relative to 8 nodes for each of the thread counts. On the right is the parallel efficiency. There is a projected 80.9% scaling efficiency for N_r=3 at 512 nodes and a projected 86.4% scaling efficiency at 256 nodes, both relative to 8 nodes.

8.3 Wire Latency

The effective end-to-end latency in the DIRT model is the sum of the latency in the NIC on the injection side, the latency on the NIC on the ejection side, the wire latency and

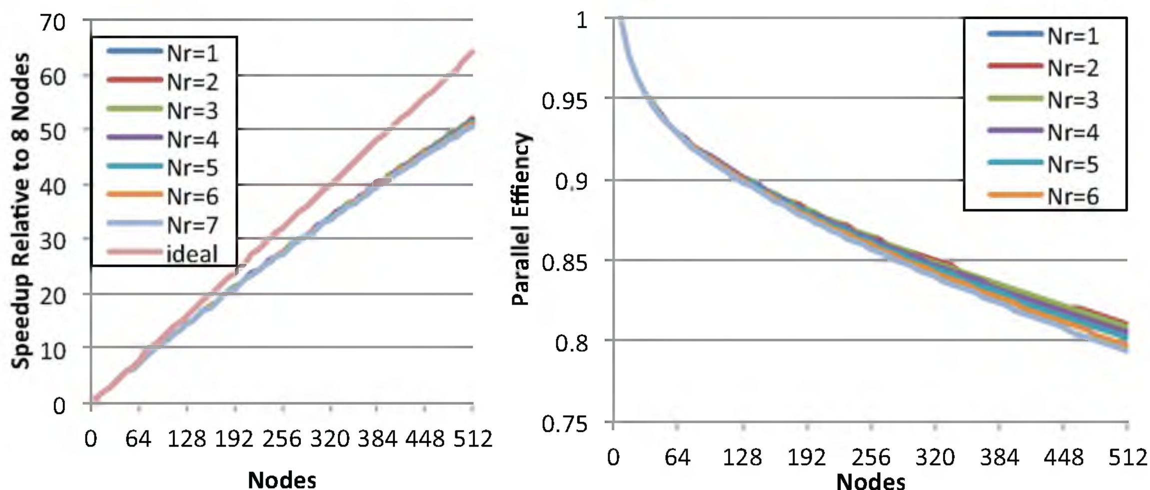

Figure 57: Scaling efficiency

serialization latency of the message. Recall that this latency is 453 ns plus 1 ns per byte. A request is 38 bytes and a response is effectively 3,350 bytes in the network including headers. The average latency between requests and responses is 2.24 µs. Figure 58 shows the effect as that one-way latency increases for each of the `Renderer` thread counts of interest. The y-axis is the effective slowdown (1/speedup) over the baseline of no additional latency. This is shown for 32 nodes.

The fact that the slope of the curve decreases with higher thread counts indicates that the additional threads are useful for hiding latency in DIRT. The N_r=3 case reaches a factor of 2 slowdown at 13.9 µs. This graph is useful for understanding the impact of endpoint and switch latency on performance. It is also useful in understanding the impact of bandwidth, which results in serialization latency. Halving the bandwidth per core to 0.5 GB/s would correspond to 2 ns per byte on the network, an average of an additional 1.78 µs per message. For the N_r=3 case that implies a 11.4% performance penalty due to the additional latency alone, not counting the additional queuing effect from the greater NIC occupancy.

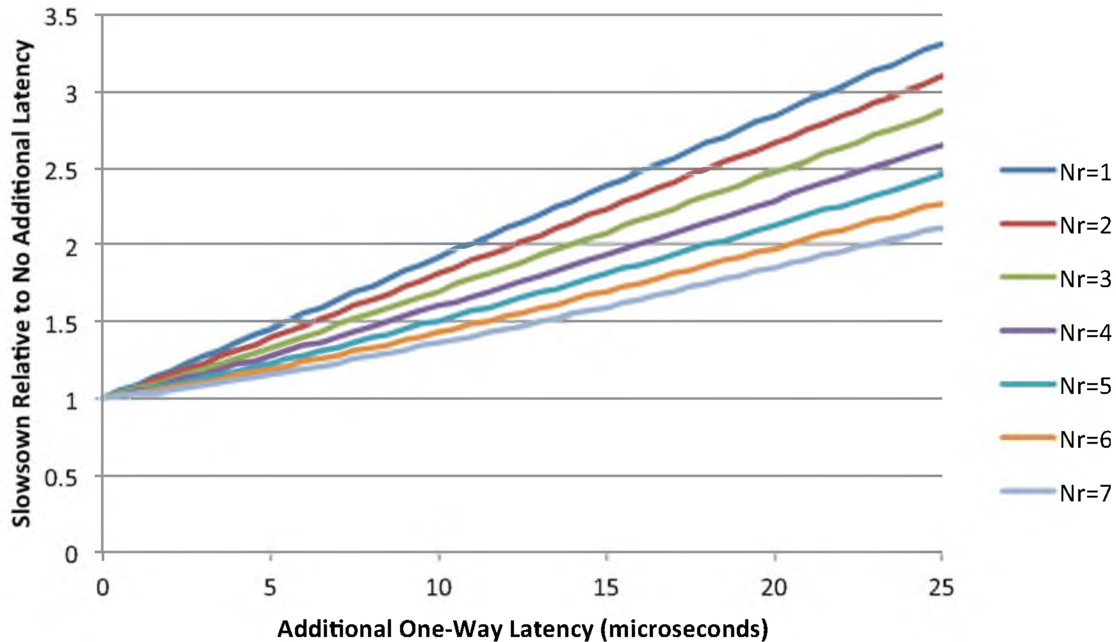

Figure 58: Slowdown on 32 nodes due to wire latency for N_r = 1 to 7

8.4 Send and Receive Overheads

Overheads can also be increased in the model to see the impact of features such as zero-copy protocols, and low-overhead interaction with the NIC. Prior work has shown that overheads are more important than latencies in application performance [79]. Figures 59 and 60 show that this is also the case for DIRT on ULN. Figure 59 shows the slowdown for each N_r as the send overhead is increased. The slowdown is a factor of 2 for an additional overhead of only 6.61 µs of additional send overhead for N_r=3. The flat portion of the curve between 0 µs and 1-2 µs of additional overhead is due to the fact that the total send overhead in that region is still less than the occupancy in the NIC due to the average message size at 1 GB/s. In that region, the additional overhead results in more contention for the SMT processor, hence a lower per-thread speedup. Beyond that, the additional overhead results in both higher contention for the SMT processor and more queuing delay.

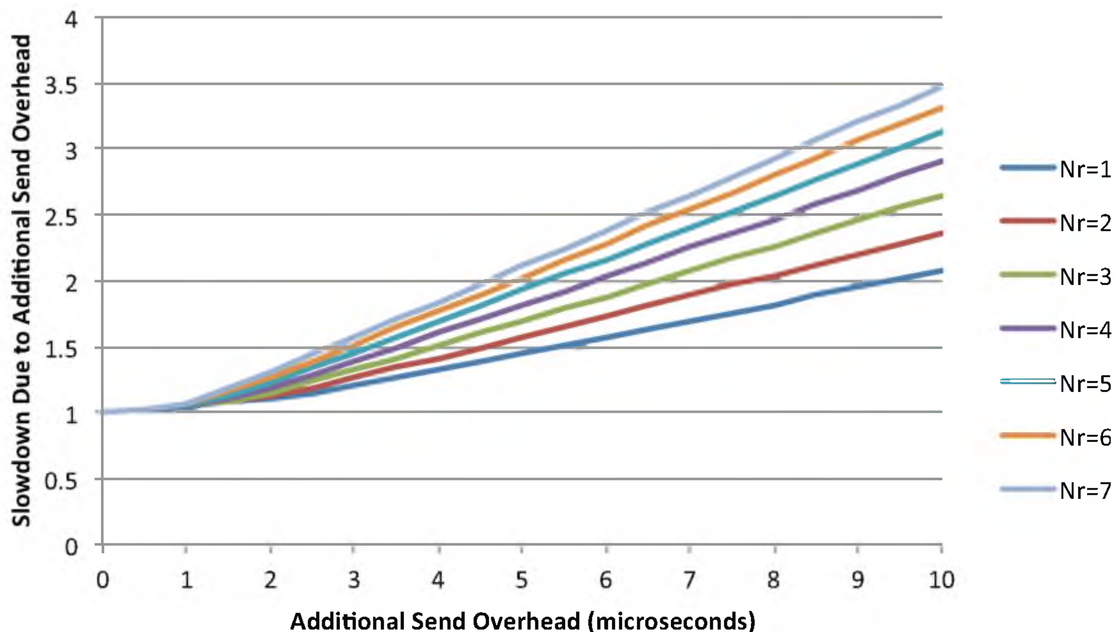

Figure 59: Slowdown on 32 nodes due to send overhead for N_r = 1 to 7

Figure 60 shows the slowdown for each N_r as the receive overhead is increased. The slowdown is a factor of 2 for an additional overhead of only 4.25 μs of additional receive overhead for N_r=3. Again, the flatter region between 0 and 1 μs is due to the fact that the total receive overhead in the `communicator` is still less than the bandwidth occupancy in the NIC.

Unlike for network latencies, send and receive overheads have a greater negative impact as N_r increases. Part of this is due to the fact that these overheads mean that the probability a thread is running increases, thus the average number of threads running simultaneously increases. This means that the average per-thread speedup decreases. Furthermore, these overheads impact the service time of the `communicator` threads. As the overheads increase, the service time increases, and along with that, the queuing delay also increases.

The receive overhead is more critical than the send overhead because there are two receives per miss in the shared handler and only one send per miss. The model is more sen-

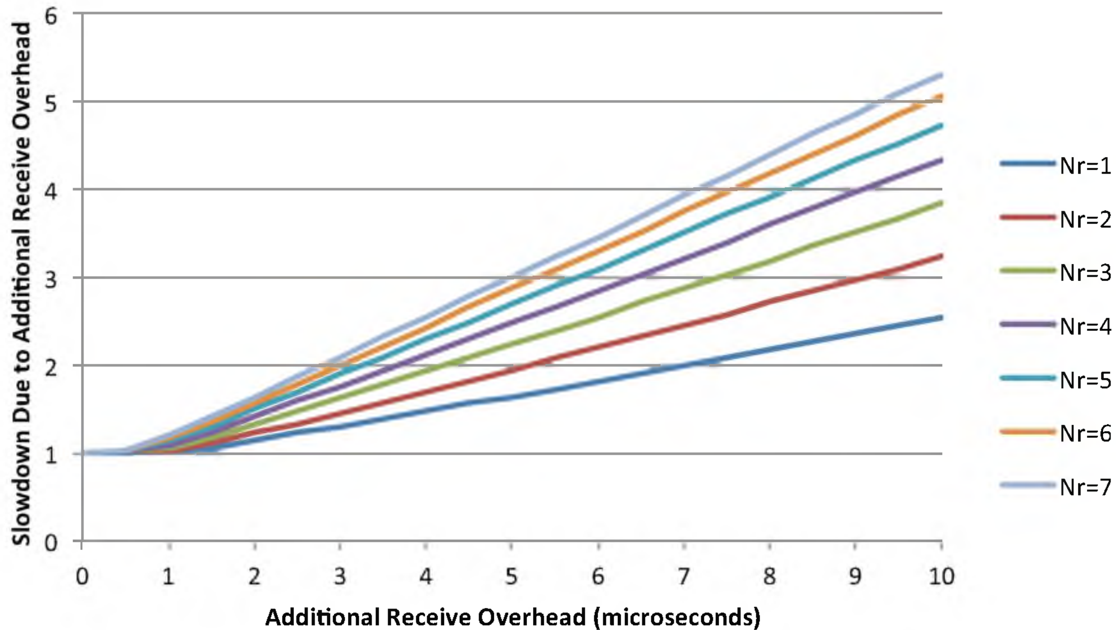

Figure 60: Slowdown on 32 nodes due to receive overhead for N_r = 1 to 7

sitive to the shared handler overhead as it has an impact on the queuing delay and it has a lower effective per-thread speedup as shown in Section 8.1. Notification overhead is a form of receive overhead, so this key result shows the importance of reducing the notification overhead, one of more significant contributions of the ULN architecture.

Injecting received messages directly into the cache reduces receive latency by approximately the amount of a cache miss for both requests and responses. The impact on the request side is modest if all of the other overheads in the recieve process are small. However, since the received scene data is consumed by the `Renderer` thread, the overhead for not placing this data in the cache behaves more like send overhead. While the impact of send overhead is less than receive overhead in absolute terms, each shared-memory fill may incur several local L2 cache misses if the data is not injected. Assuming one-third of the data is referenced, this can result in up to 800 ns of overhead at 50 ns overhead per L2 miss.

Comparing the ULN architecture to an architecture with all of the other features except user-level notifications can be done by looking at Figure 59. An optimistic overhead of 1 μs per notification corresponds to a slowdown of 1.053 for 32 nodes and N_r=3. Having 2 μs of overhead per notification results in an overhead of 31.9% over ULN. In other words, having the user-level notifications in this architecture results in a 5.29-31.9% speedup over an architecture without user-level notifications. Figure 61 shows that the penalty of the additional receive notification overhead decreases slightly, going from 5.31% down to 5.03%, as the node count increases. However, for the 2 μs overhead, the benefit increases from a 31.7% improvement at 8 nodes to 33.5% at 256 nodes.

This is a good speedup on a problem where the message size is not fully optimized for the architecture and thus causes the system to be bandwidth constrained. The benefit of notifications where the message size has been optimized is discussed in Section 8.9.

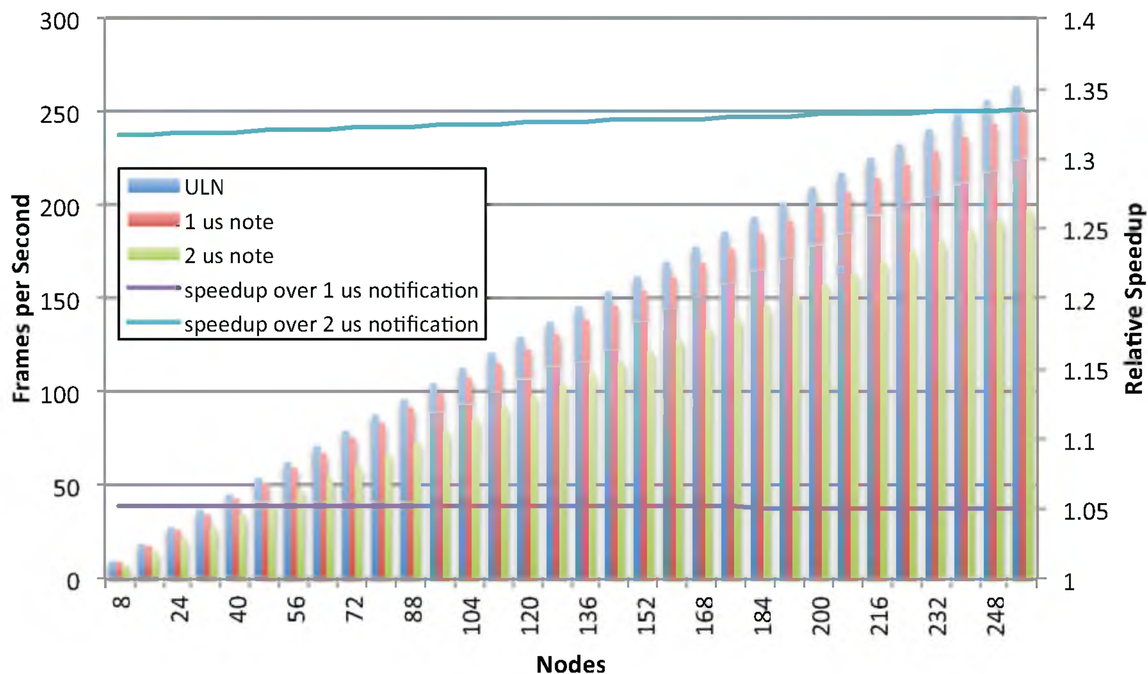

Figure 61: Slowdown due to 1-2 microsecond notification overhead, N_r = 3

Comparing ULN to an efficient OS-based NIC requires having overheads for sends, receives, and notifications. Figure 62 illustrates the speedup of the ULN architecture over a SMT-based architecture that does not have user-level notifications and requires OS or other high-overhead interaction to interact with the NIC. The overheads modeled are 1 μs for send overhead, 1 μs for receive overhead, and 1 μs for notification overhead at N_r=3. As above, the speedup of ULN over the baseline architecture improves as the system size scales. At 32 nodes, ULN is 50.9% faster than a traditional OS-based NIC under these conditions.

8.5 Thread Communications

Polling for message receives in the ULN architecture would result in all of the threads competing for the processor all of the time instead of just when there is useful rendering work or miss handling work to perform. At 32 nodes, for N_r=3, there are 4 total

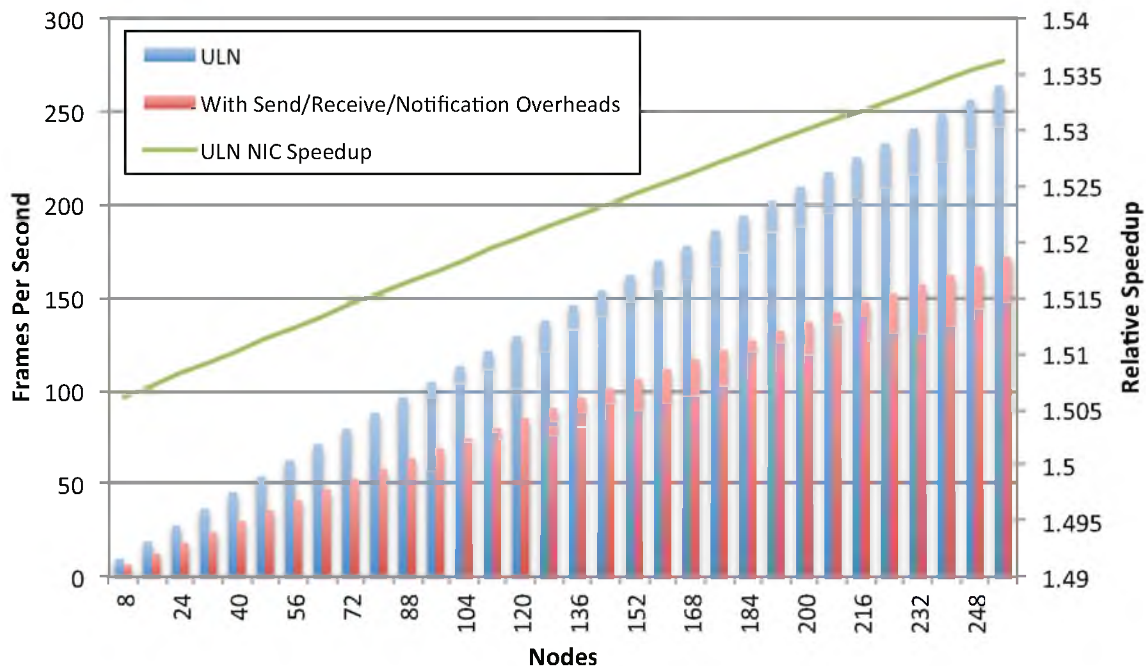

Figure 62: Speedup of ULN versus OS-based NIC, N_r = 3

threads, and S_h is 0.8066. The expected speedup for 4 threads running simultaneously according to the model in Section 4.4 is 0.621. This speedup does not apply to the latency portion of the communication. Since the computation accounts for 68.5% of the 32 node, N_r=3 time per miss, the performance penalty for polling full time would be 20.5%

By adding overhead to both the `Renderer` thread and to the `communicator` thread we can get an approximation of how overheads in the interthread communication system affect performance. Since there is only one thread wake per communication, adding just the wake overhead to the `communicator` has one half the impact of receive overhead. Adding the same amount of wake overhead to both the `communicator` and the `Renderer` threads has the same impact as adding send overhead. Figure 63 shows two views of the same data representing the speedup of a light-weight thread wake mechanism versus a lock-based mechanism with OS overheads. The wake overhead is the overhead in the `communicator` thread required to wake the appropriate `Renderer` thread. The T_r overhead is the overhead in the `Renderer` thread generally paid when the thread sleeps.

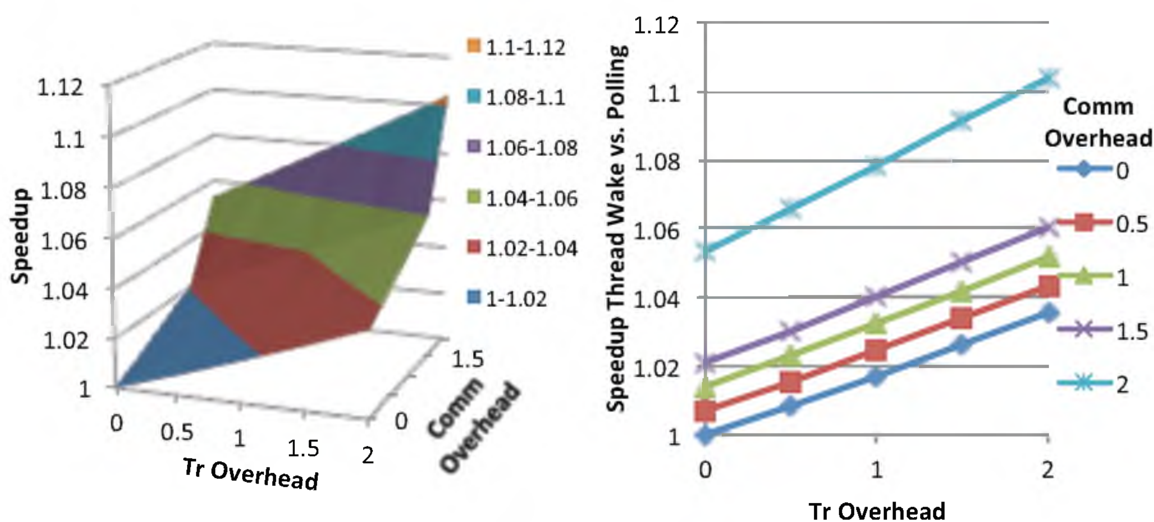

Figure 63: Speedup of ULN thread wake versus OS-based lock, N_r = 3, 32 nodes

On the `Renderer` side, just adding T_r overhead to model wake overhead would result in an over-estimation of the impact due to the way the model works. Adding this not only models the overhead of extra time spent with the `Renderer` thread running before sleeping, but also adds effective end-to-end latency to the message. Since the thread sleep happens after sending the cache miss request, that additional latency is not accurate. This effect is compensated for by reducing the round-trip wire latency by an equivalent amount.

This data makes clear that the wake overhead in `communicator` is more critical to performance than the sleep overhead in the `Renderer`, since a 2 μs of wake overhead results in a performance penalty of 5.29% versus a 3.52% penalty for a 2 μs sleep overhead. This is consistent with the observations made above that overhead in the `communicator` is more critical than overhead in the `Renderer` above.

A reasonable overhead for a sleeping lock operation on the `Renderer` thread is on the order of 1-1.5 μs. A reasonable trip into the OS in the `communicator` thread to wake the `Renderer` thread is on the order of 0.5-1 μs. This implies that there is a 2.45% to 4.19% speedup on ULN compared to an architecture without an efficient thread synchronization table. Having overheads as low as 5 ns is probably unnecessary, as there is a 0.83% performance penalty for adding 500 ns to the sleep side and a 0.69% penalty for adding the same to the wake side. Adding 50 ns to both of those mechanisms would result in less than a 1% penalty. However, having a low overhead mechanism is not costly given that the synchronization table is needed for the notification mechanisms. Plus, these overheads become more significant as communication becomes more frequent.

8.6 DMA versus PIO

DMA sends in the ULN architecture have such low overhead that they are more efficient than PIO messages in all but the tiniest of messages. Even sending the 20 user-byte send request using PIO with an extremely efficient implementation will reduce T_r by 18 ns from about 15.288 μs to 15.270 μs and will reduce the message latency on the request side by 10 ns. The result would be a 0.12% performance increase. The reason for supporting PIO in the ULN architecture is for software convenience rather than for performance.

For the responses, the data for the response is already in memory. This data would have to be read in and written to the NIC. Assuming the `communicator` thread consumed half of the load/store bandwidth of the processor without impacting the performance of the `Renderer` threads, there would be a 3.03% performance penalty over the DMA method at 32 nodes and N_r=3. Assuming the communicator could only get one-quarter of the load/store bandwidth, the performance penalty would be 12.8%.

Neither of these are huge penalties, as the SMT processor can perform those operations efficiently while allowing other work to proceed concurrently. However, the cost of providing DMA in a modern NIC architecture is very small. For a different style of code where the compute thread could continue computation after a send, the DMA features would have an even greater impact.

In addition to the performance benefits, there would be a power benefit of not bringing cacheline all the way to the processor only to send them back down to the NIC. Finally, as the DMA transfers are so efficient even for the smallest of messages, one could consider changing the maximum PIO transfer to be much smaller, possibly even only a single cacheline. This could simplify the implementation of the NIC, allowing it to restrict PIO to single

transaction operations. This would still allow PIO operation to still be used for efficient control messages and would allow for an efficient mechanism for supporting partitioned global address space puts. Larger transfers could be done as either a series of small PIO messages or as a lightweight DMA operation.

8.7 ULN Performance Summary

The preceding sections characterize the benefits of the various features of the ULN architecture. The model developed in Chapter 7 proves very useful in evaluating a wide spectrum of configurations and options. The ULN architecture provides a 50.9% performance improvement over a more traditional OS-based NIC and a 5.29-31.9%% improvement over a user-level NIC due to the user-level notifications alone.

Figure 64 compares the performance impact of latency, receive overhead, send overhead, and notification overhead for the 32 node, N_r=3 case. The impact of receive overhead is identical to notification overhead, and thus is hidded behind the notification overhead curve in the graph. The slopes of these curves can be used to understand the relative contributions of each part of the architecture when combined. Notification overhead is estimated to be about 2 times recieve and send overhead in modern architectures, and the improvements to latency are less than half that of the other improvements. Under these assumptions, the reduced latency contributes, receive overhead reduction, send overhead reduction, and notification overhead reduction contribute 18%, 16%, 35%, 32%, respectively, to the projected speedup of ULN when those latencies and overheads are small and 3%, 27%, 16%, and 54%, respectively, as those overheads approach the right half of this graph.

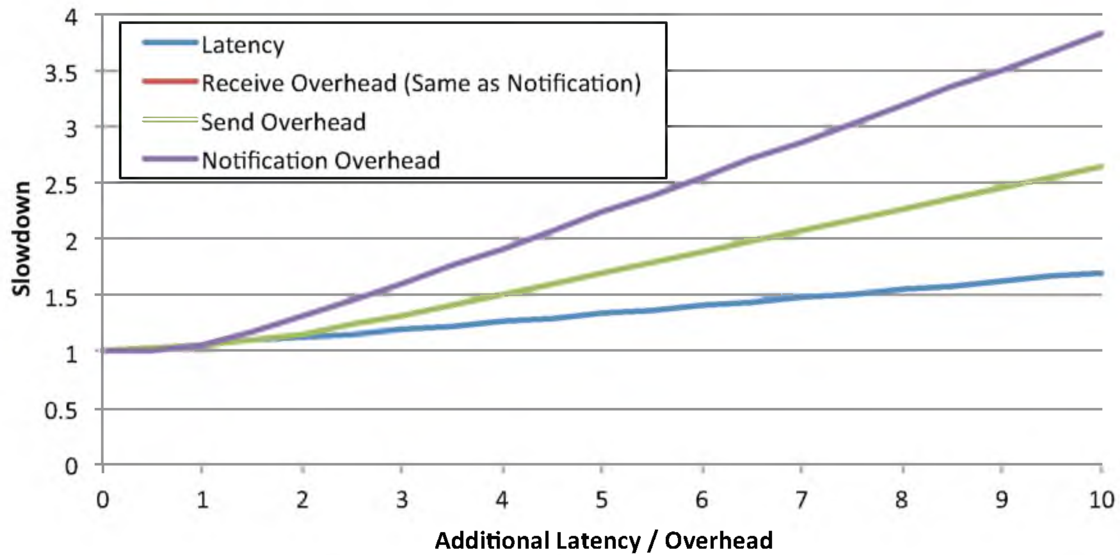

Figure 64: Impact of additional latency and overheads

In addition to showing the benefits of the ULN architecture on the existing DIRT code, the model can be used to investigate possible enhancements in DIRT supported by the architecture. Two of these what-if scenarios are discussed in Sections 8.8 and 8.9.

8.8 What-if: Finer-grained Cache Block

When a miss occurs in the dataset being visualized, there is a trade-off between fetching a large block of data to fill the local cache, thus minimizing overhead per byte transferred, and having a small block reducing over-fetch, thus moving unnecessary data and wasting space in the cache. In the cluster platform that DIRT was designed and evaluated on, the messaging overheads are significantly higher than on ULN. This lead to a large cache block being chosen to amortize large message overheads per cache miss request. Recall that the data points in the volume being visualized are 16-bit (2-byte) short integer values. A cache block is organized as a 12x12x12 subvolume of these short values, resulting in a 3,456 byte block. Consider a ray passing through this subvolume normal to one of the

faces, perfectly aligned with the voxel, the ray will trace through 12 elements or until an isosurface match occurs, whichever comes first. A set of rays from the same assignment will reuse the nearby values in the same block. Rays pass through this subvolume in a random orientation, however, the same basic operation applies.

A simple model for reducing the block size would be to imagine using a 6x6x6 subvolume. This subvolume is one-eigth the size of the 12x12x12 block. The worst case would be that a miss of a single 12x12x12 block in the original would result in 8 misses if a 6x6x6 block were used. However, the expected miss count grows approximately with the area of the face of the block versus the volume of the block. This implies that the expected miss rate grows closer to a factor of 4 while the block size reduces by a factor of 8. There would also be an additional benefit due to a finer-grained mapping into the acceleration structure at the cost of slightly more memory consumed by that structure.

Other block sizes between these two values are possible and reasonable, including square aspect ratios, such as 9x9x9 and rectangular aspect ratios such as 9x9x10. The expected cache misses for a square aspect ratio follows a squared-cube law, where the block size shrinks by a factor of n and the cache miss rate increases by approximately $n^{2/3}$.

Figure 65 shows a family of curves for the expected performance, using the squared-cube law, for a block sized nxnxn on 32 through 64 nodes for N_r=3. The performance above corresponds to a 12x12x12 or 3,456 byte block size.

The optimal cacheline size according to this study for all node counts from 24 nodes to 256 nodes is a 7x7x7 element block. This corresponds to a 686 byte line. For node counts 8 and 16, the 8x8x8 block is slightly better than a 7x7x7 block, by under 0.005%. The worst case difference between the optimal 7x7x7 block and the 8x8x8 block occurs at 256 nodes

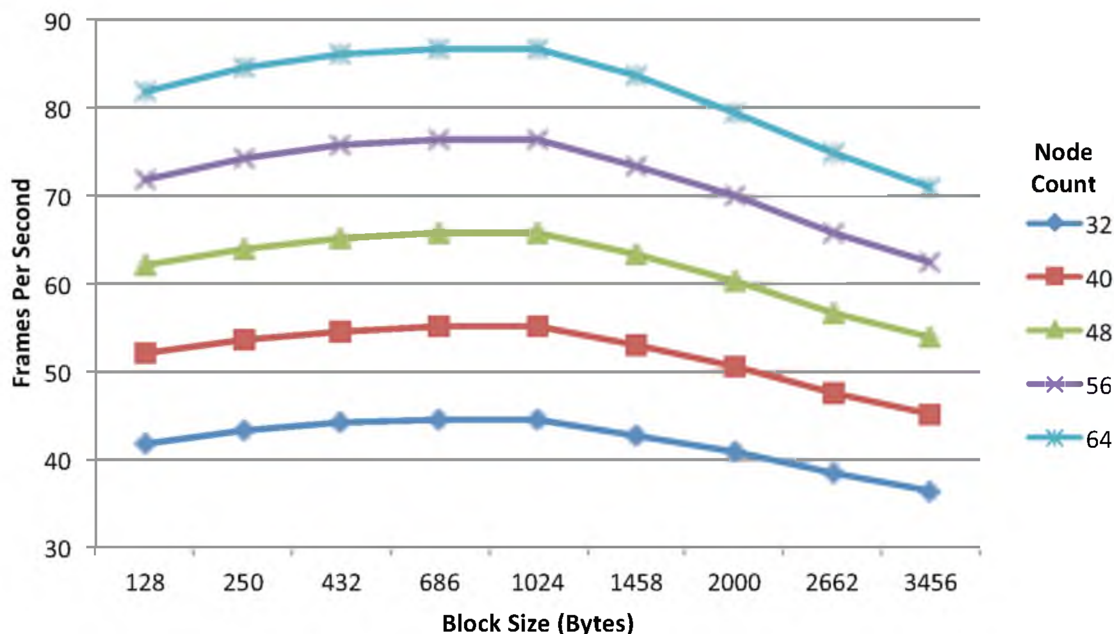

Figure 65: Optimal cacheline size, expected cache-miss behavior

and is only 0.0076%. Furthermore, the 7x7x7 block is better than a 6x6x6 block by only 0.74% at 8 nodes to 0.84% at 256 nodes. For smaller cache blocks, the performance drops fairly gently. For larger cache blocks, the time to send the response due to the bandwidth constraint exceeds the time for the `communicator` to handle an incoming message. Thus the handling time becomes NIC bandwidth bound and the performance drops quickly. This indicates it is better to err on the side of a smaller cacheline size.

Figure 66 shows a the corresponding family of curves for the worst-case performance of the miss rate increasing with the ratio of the block size decrease. The worst-case is calculated by multiplying the misses by the ratio of the block size to a 12x12x12 block. Under that condition, the optimal cacheline size is 8x8x8 for all node counts. The fall-off is a little more steep on both sides, with a 0.97% to a 0.99% performance penalty for moving to a 7x7x7 cache block.

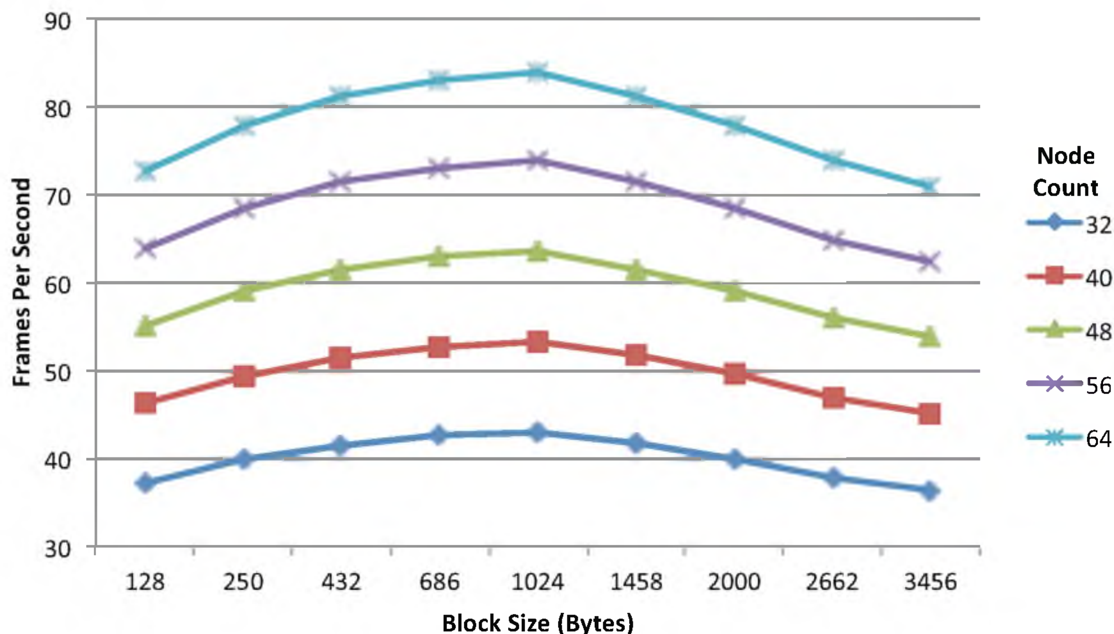

Figure 66: Optimal cacheline size, worst-case cache-miss behavior

The best case would be one in which the number of cache misses does not increase as the line size decreases. Clearly this case favors the smallest cacheline size possible. Figure 67 directly compares the worst, expected, and best case for 32 nodes.

Again, it is clear that the fine-grained message features favor a smaller cache block than the point chosen for DIRT on the cluster. The optimal point results in a block size that is a little under one-fifth of the original block size and a block size one-eighth of the original are still in the range of optimal. This is an interesting result as it indicates that optimizing for finer-grained messages not only impacts performance directly, but also allows for software to optimize for a more natural point, gaining additional performance by doing so.

8.9 What-if: Directly Resume `Renderer`

In ULN a single thread is commonly used as a received communications handler. This is how DIRT is modeled. However, the architecture makes it possible to have any

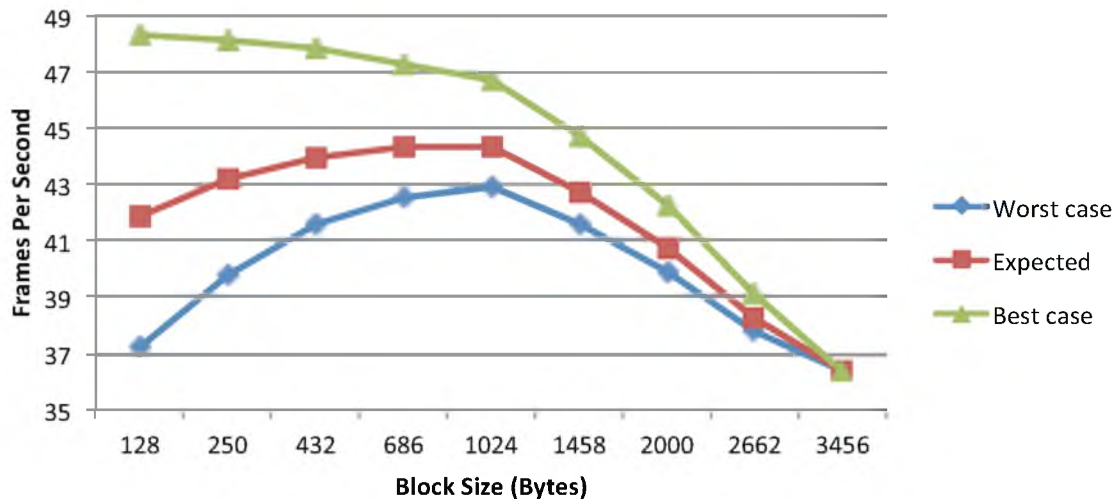

Figure 67: Comparison of worst versus expected versus best case

thread block on different communication events. It would be simple to direct requests to the `communicator` thread and to direct a response to return directly to the original requesting `Renderer` thread. The changes to DIRT required to support this are minor. A requesting `Renderer` thread would have to include its context ID in the request and the `communicator` thread would have to respond, targeting the supplied context ID instead of the one associated with the source node's `communicator` thread. Finally, the `Renderer` thread would have to perform the cache functions of marking the line valid and waking any other subsequent `Renderer` threads that waited on the same line.

This optimization can be easily modeled with the model above by estimating the overhead of the `communicator` on the response versus on the request and subtracting the appropriate overhead from the `communicator` and adding it to the `Renderer`. From Table 8, the time spent in `semaphore.get_sole_access` is associated with miss responses. Outside of that function, from inspection of the code, many of the code paths are shared or similar. Thus, it is assumed that the time spent processing in the `dataserver_group::handlemessage` and `dataserver_direct::han-`

`dlemessage` functions is split evenly between request and response processing. Thus, 56.0% of 0.677 μs spent in the `communicator` per miss is associated with miss response handling. Additionally, there is another 32 ns associated with receiving the response and 2 ns of overhead associated with the wake operation. The resulting overhead of 0.413 μs is subtracted from T_h. The 5 ns thread wake latency is also removed from the effective round-trip time.

Conservatively all of the overhead subtracted from T_h except the wake overhead is added to T_r. However, it should be noted that the purpose of the `dataserver_group::handlemessage` function is to select the appropriate dataserver based on the request. Since the `Renderer` thread already calls the associated miss function and the response is sent directly to the requestor, that selection is redundant. Thus, in the aggressive case, we only add 38.1% of the 0.677 μs plus the receive overhead, for a total of 0.293 μs to T_r.

For the default block size of 3,567 bytes, the conservative optimization results in a 1.31% to 1.37% performance penalty in the range of 8 to 256 nodes. Even the more aggressive optimization results in a slight 0.77% to 0.81% performance loss. The reason for this is that with those large blocks, the T_h overhead in the model is effectively hidden under the serialization latency of the large block message. As that overhead is moved to the T_r side, the overhead is fully exposed.

However, as the block size decreases, the optimization begins to make a small difference. At the optimal 7x7x7 block size from Section 8.8 of 686 bytes, there is negligible (0.344% to 0.397%) performance gain for the conservative approach at N_r=3, and for the aggressive approach a small (1.50% to 1.64%) gain. However, the reduced overhead in the `communicator` makes the optimal block size in this case a 6x6x6, 432 byte block. Figure

68 shows the comparison of this optimized code at varied block sizes compared to the traditional response handling for a fixed best-case 7x7x7 block size of 686 bytes.

This shows that there is a small but nonnegligible 3.28% to 3.70% additional performance to be gained by further optimizing the application in this way for the ULN architecture. Since the performance gain is small and only occurs for a narrow range of block sizes, it is important understand whether this benefit makes the architecture more sensitive to the optimal block size. Figure 69 shows the speedup of the optimized versions of the code versus the baseline as the block-size is varied for both the optimized and unoptimized code.

The optimized code improves the performance in the 4.81% to 16.6% range over the unoptimized code for smaller than optimal block sizes. The average improvement over this range is 8.99%. Performance for larger than optimal block-sizes ranges from a 1.14% penalty to a 1.15% improvement with an average penalty of 0.55%.

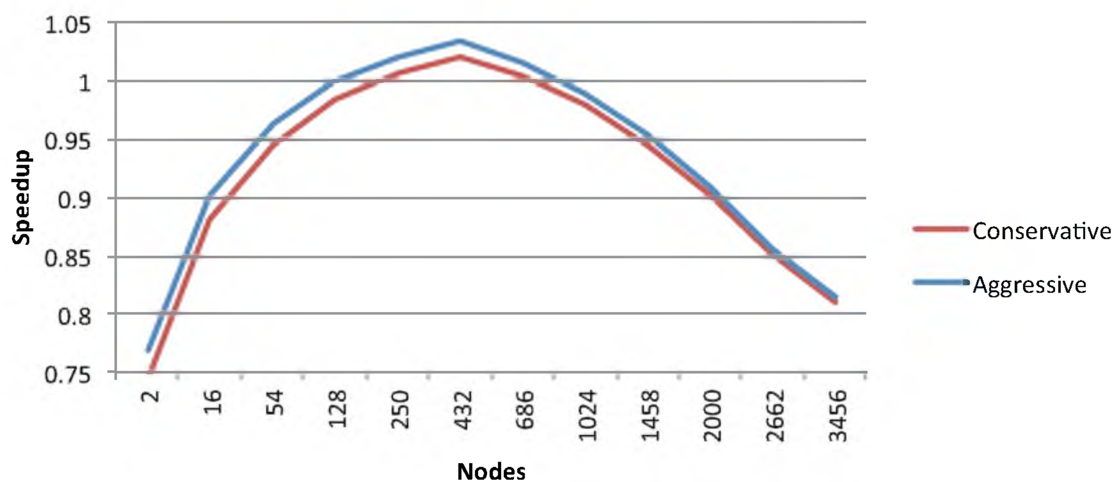

Figure 68: Speedup of optimized response returns versus best-case block without

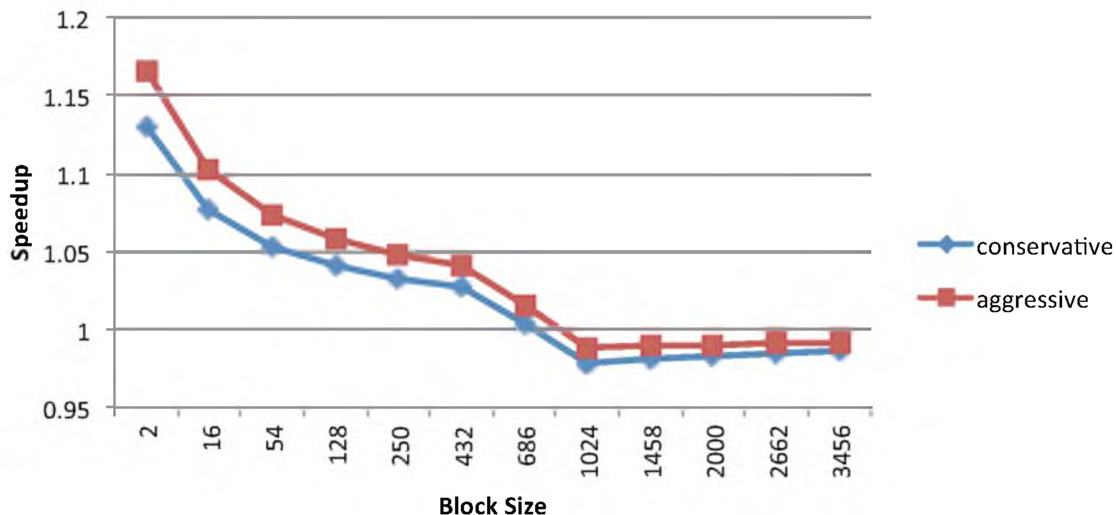

Figure 69: Speedup of optimized code over best-case 7x7x7 block

8.10 Combined Benefits

The combination of the ULN architecture and these what-if optimizations results in a speedup that ranges from 1.89 to 1.99 over that of the original code running on an OS-based NIC with send, receive, and notification overheads of 1 μs each. Not only does the combination of ULN and the optimizations it enables provide a good speedup, but it improves the overall scalability of the code. The speedup of ULN with the additional proposed optimizations is shown in Figure 70.

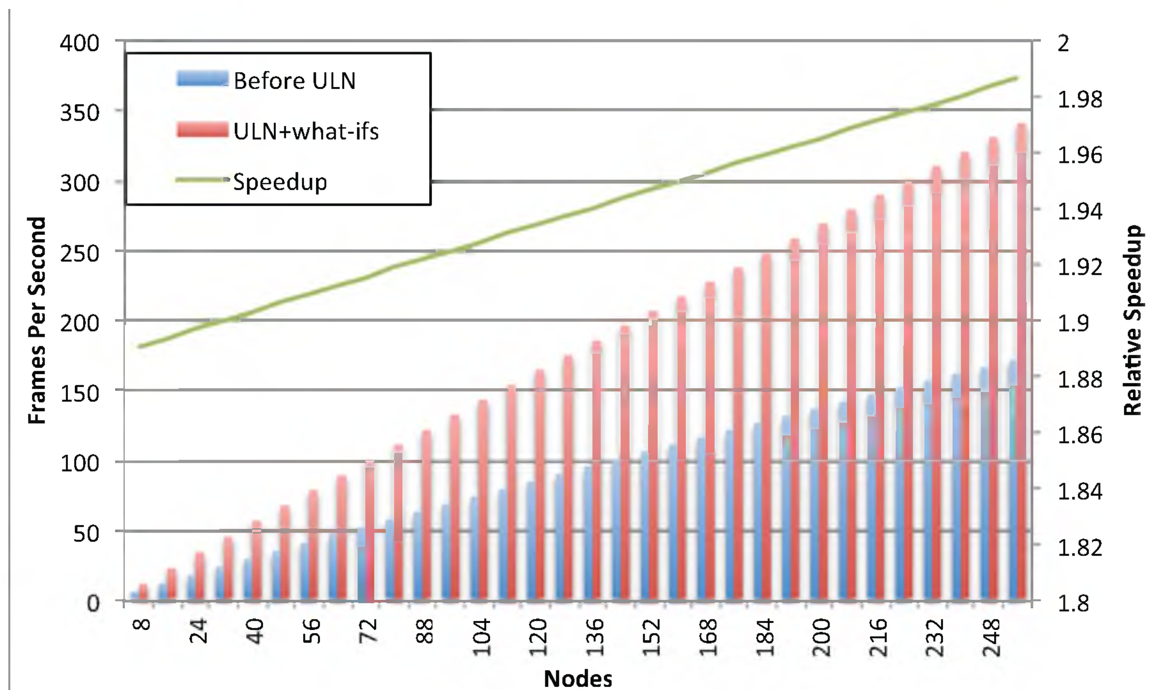

Figure 70: Speedup of ULN plus what-if optimizations, $N_r = 3$

CHAPTER 9

RELATED WORK

There is a rich set of related work that has preceded ULN, related to each of the key architectural aspects. This includes work on multithreaded processors, thread synchronization, efficient message passing protocols, tight NIC integration, and even several proposals for complete message passing architectures. This chapter addresses related work in each of those areas in turn.

9.1 Simultaneous Multithreading

Simultaneous MultiThreading (SMT) architectures [40,75,108,109,110,111], as proposed by Susan Eggers et al. from the University of Washington, have seen commercial success in the Intel Pentium4, Intel Core i7, Intel Atom chips [66], and IBM Power5, 6 and 7 [57] architectures. SMT architectures combine the benefits of traditional superscalar architectures with that of more recent multithreaded architectures to provide a processor core that is capable of issuing instructions from multiple threads across multiple execution units on a single cycle. This is accomplished in part by duplicating context registers and associated context handling hardware to allow multiple contexts to be active within the CPU at the same time. In short, SMT architectures provide the ability to simultaneously take advantage of both instruction-level parallelism (ILP) and thread-level parallelism (TLP) within a single processor core.

Having an SMT processor provides benefit in addition to just enabling the user-level notification mechanism discussed in this work. The SMT processor is targeted to tolerate latency and hide overhead by allowing one thread to process overhead or wait for long latency operations, such as memory access or synchronization while the execution of independent threads continues. When one thread is stalled on a long-latency network operation, other threads can still utilize the available functional units. Since multiple threads can be active on any given cycle, overhead processing can execute in parallel with primary computation. In particular, the SMT processor allows communications threads to run concurrent with computational threads sharing unused instruction issue slots in the processor core. Thus, overhead tasks related to communication can be at least partially hidden underneath computation tasks.

9.2 Thread Synchronization

Dean Tullsen et al. [110] show how extra lock and release hardware can be introduced to provide fine-grained synchronization for threads in an SMT processor. This efficient locking mechanism allows one thread to block on a hardware semaphore and be released by another co-operating thread quite efficiently. A blocked thread does not consume any processor issue slots, allowing other threads to proceed as usual. Thus, a blocked process does not introduce any extra overhead, other than that of holding a context control block. However, the thread synchronization hardware proposed in Tullsen's work is focused on threads communicating within the subsystem of a single core.

The notification primitive in this work extends some of the control over this hardware locking table to external events, such as message arrival notifications from the NI. In this way, asynchronous I/O events can be delivered directly to the appropriate user thread

without the help of a privileged kernel. This same mechanism can be used to help the processor tolerate message latency by freeing the CPU to process other threads instead of polling or requesting an interrupt for a message arrival. In effect, this work pushes the SMT idea one step farther by extending many of the mechanisms to the I/O realm.

9.3 Efficient Protocols

A number of message-passing protocols have been proposed. The following does not represent a complete list of those protocols, but instead compares those which are most relevant to this work.

Wilkes' Hamlyn [20] presents sender-based protocols, to reduce overhead. Having the sender manage its destination buffers implies that data can be easily and effectively received directly into the receiver's process space while simultaneously side-stepping the issues of buffer-overruns. The Hamlyn network interface, though designed for closely-coupled clusters, resides on an I/O bus. This again implies extra overhead, latency, and bandwidth limitations. Avalanche [106] also used a sender-based messaging protocol, called the "Direct Deposit Protocol" (DDP) [107], and a local message cache to reduce latency and overhead.

Sender-based protocols allow simple and efficient hardware to place incoming messages directly into the receive process's address space. In sender based protocols, the remote node manages the local receive buffer, and places incoming messages directly in the appropriate location in memory, which avoids unnecessary copies. This avoids kernel involvement on receives. The message cache stores incoming messages, which saves the extra system bus traffic of writing the messages to main memory, and provides for quick

retrieval on a processor request. These two ideas will also be used in the evaluation of the proposed work.

Unlike Hamlyn and DDP, the protocol in this work uses virtual addresses in combination with an NI TLB to remove restrictions on receive buffers.

Thorson von Eicken's U-Net [41,114,115] reduces communication overhead and latency by virtualizing the network interface. In one implementation, Cornell uses an off the shelf I/O bus network interface, which contains an embedded processor to provide a safe, user-accessible communication medium. In other implementations, where the local interface card does not have any on-board processing or user-level protection capability, the direct interface is faked by the local kernel. The local process communicates with the network interface by placing and picking up message packets from per-process send and receive queues. In earlier implementations, send and receive data must be placed into pinned pages, to avoid page eviction problems. In a later proposals, Welsh suggests placing a TLB in the network interface to avoid the added restriction of fixed pinned pages [115]. The architecture in this work also gives the NI access to a TLB to allow sends and receives to be handled in user-space.

However, the network interface in this architecture differs from U-Net in that NI is placed on the same die as the CPU and is tightly integrated with the CPU core. This limits flexibility in terms of being able to adapt to a new network by simply removing a network card and plugging in another. It does, however, provide for much lower latency, it allows a much higher bandwidth, and it enables efficient notification mechanisms, all of which are necessary for a fine-grained communication multiprocessor.

Active Messages [42,76] embeds a message handler in the header of a message. When a message arrives, the message handler is executed to handle the payload of the message. This enables a custom mechanism to be implemented to handle each and every message arriving at a node. Though it is not specifically a goal of this work, it has been noted that this architecture would provide a good hardware base to implement Active Messages. A thread waiting for a message could immediately jump to the handler code in the header without the penalty of an interrupt and without interfering with the currently running thread. If messages are received directly into a message cache, then this handler code could potentially execute directly out of the message cache, also saving the cache overhead of bringing in conventional handler code. Illinois' Fast Messages [58,59,70,88,89] is effectively a platform independent implementation of Active Messages.

The Cray T3E [100], X1 [3,39], BlackWidow [2], XMT [64,82], and XE6 [9] architectures all provide hardware mechanisms for Partitioned Global Address Space (PGAS) that makes all of the memory in the machine accessible by all of the processors. The X1, BlackWidow, and XMT architectures provide direct load and store access to all of the memory in the machine by translating a 64-bit virtual address to a global physical address composed of a node and a physical address within that node. The T3E and XE6 architectures provide this by providing windows of local addresses that map to global addresses. In all of these machines, only local node addresses are cached. References to remote node addresses are not placed in a local cache. Global memory references are kept coherent by maintaining coherence within the home node and disallowing that data to be cached elsewhere. This model behaves much like the sender-based protocol in the ULN architecture in that writes, or in the ULN case, message sends, may target all of the global

memory. It also adds to that the ability to read a snapshot of remote memory. Adding this mechanism to the ULN architecture should be investigated in future work.

Work has also been done to optimize message-passing protocols that are built on top of shared memory systems [23,46]. ULN presents a flat message-passing architecture, both on-chip and off-chip, thus these optimizations are not directly applicable to ULN. However, it would possible to combine a traditional shared memory model on-chip with an off-chip message passing architecture based on ULN. Future work should consider such an architecture.

9.4 Message Passing Architectures

Several previous systems have advocated moving the NI closer to the CPU. Flash [67], Avalanche [106], Alewife [5], SHRIMP [12], and Tempest [97] all placed the NI directly on the system memory bus. Having the NI attached to the system bus significantly reduces the overhead of accessing it compared to having it attached to an I/O bus. In addition to reducing overhead, placing the NI on the system bus allows these systems efficient access to coherency traffic, which several of these systems use to an additional advantage. The MIT J-Machine [31] and M-Machine [44] take it one step closer by bringing the NI directly onto the custom processor. Alewife, the J-Machine, and the M-Machine also have interesting characteristics in common. They all use a thread model or a thread-like model to deal with communication. Alewife uses a modified SPARC processor in an unconventional way to implement these threads.The J-Machine and M-Machine both build a custom processor to get the desired thread behavior. SMT processors now seem to be a natural way to achieve effective functionality of these machines with only minor modifications to the CPU structure.

In many ways the ULN architecture shares the same goals as the M-Machine. Both take advantage of thread capable processors to hide message overhead, and both have forms of automatically dispatching threads when a message arrives. The ULN architecture differs from the M-machine in the following ways. Messages are received directly into a user's address space with hardware support, eliminating the need for trusted message handlers. Incoming messages are placed into a message buffer, or message cache, to avoid pollution of the processor's cache hierarchy. Finally this architecture is based on modifications to existing SMT architectures.

This work extends the Avalanche Scalable Parallel Processor architecture developed at the University of Utah. Avalanche [33,106] placed the network interface on the system bus, keeping it close to the processor. This allowed it to participate in coherency traffic, and thus maintain a local network cache. The local cache enables the Avalanche network interface, named the "Widget," to supply network data to the processor more quickly than the main memory. In addition, it avoids the overhead of wasting system bus bandwidth to transfer message data across the system bus twice. Namely, on the way to main memory on message arrival, and on the way back to the processor when the message is consumed.

The work builds on Avalanche by eliminating kernel involvement on message sends and message arrival notification. User processes are given direct access to the NI's send engine. Security is maintained via an NI TLB. The user process communicates virtual addresses to the network interface, which are checked and translated to physical addresses using this TLB. In this way, the NI can both ensure that the user-process is requesting sends of legal memory and perform the address translation service for the user process.

The MIT Alewife [4,5,6,7] is a hybrid message-passing and DSM machine that employs several mechanisms to reduce communication latency and overhead. The network interface is effectively coupled to a SPARC based CPU core at the cache interface level. The integrated Communications and Memory Management Unit (CMMU) serves as the cache controller, network interface, and main memory controller. This design afforded Alewife the flexibility of more tightly coupling the network interface with the processor core, without having to squeeze both on a single die.

The Alewife CPU core is a modified SPARC-based processor, called Sparcle. Sparcle was augmented with several custom instructions to more directly and efficiently handle sending and receiving of messages. Hardware support and software conventions are used to implement a block multithreading scheme using the SPARC's register windows. Instead of using the windows as an efficient mechanism for function calls, Alewife uses these register windows as separate thread contexts.

By convention, one of the contexts is dedicated to exception handling. This context is primarily used to avoid a context switch when a message arrives. As messages arrive, they commonly trigger an exception. Special interrupt lines are brought out of the Sparcle ASIC, which signals a message arrival. This style of interrupt causes the exception context to quickly vector to a message handler.

The Sparcle processor also has a few extra instructions that provide for efficient context switches. On a trap, the Sparcle can switch to a new context in a mere 14 clock cycles. A process can block on a remote access incurring only a few cycles of overhead and switch to a new thread while the remote access is outstanding.

The MIT J-Machine [29,30,31,87,103] reduces send/receive overhead by tightly coupling the network interface with a custom processor, the Message Driven Processor (MDP). The entire processor and instruction set is designed around the idea of sending, processing, and receiving object-oriented messages. Messages arriving at the MDP are handled immediately if the processor is ready to consume the message. If not, they are buffered in a message queue in local memory by the Message Unit (MU) for subsequent handling. The overhead of sending and receiving message is on the order of tens of clock cycles. Protection is provided by only allowing a single user process on the machine at a time. A general purpose host CPU is used as a front end interface to access the parallel capabilities of the machine.

The J-Machine uses a data-flow style processor with two register sets to efficiently handle message arrival. If an incoming message has a higher priority than the currently executing message, the other registers are used to begin execution on the incoming message immediately, without buffering the message in a message queue. Incoming messages specify how they are to be handled. As messages are handled, a handler class specified in the message header was invoked by looking up the handler in a table. A message handler could begin execution of a message within a few cycles of the arrival of a message header if the processor was idle or executing a lower priority message.

The J-Machine provided the user with two sets of registers, a high priority and a low priority set. Arriving messages can have either a low or a high priority. If the priority of an incoming message is equal to or greater than the currently executing process, the incoming message uses the register set corresponding to its priority, and begins execution immedi-

ately. Otherwise, the message waits for the processor to switch back to a lower priority before being consumed.

The MIT M-Machine [44,60,71] is made from custom, multithread capable, integrated CPU and network chips. Each node contains 12 function units local cache and a local memory controller. The entire cluster maintains a flat address space, accessible from any processor. The M-Machine, unlike its predecessor, adds safety mechanisms enabling easier debugging, and multiprogram support.

This work differs from the three MIT machines in that it uses a simultaneous multithreading processor to provide latency tolerance and concurrent handling of message arrival events. This allows it to achieve many of the benefits of the MIT machines in a modern processor and also reduce the cost of blocking on message arrival events. The work also provides for a multiuser model, and does not require that the entire machine resources be dedicated to a single user at any given time.

The Stanford FLASH machine [48,49,67] integrates the network interface with the memory controller. This is done primarily because it is intended to be more of a DSM machine. However, integrating the network tightly with the memory controller affords that FLASH is able to see and participate in coherency traffic. In this way, as with other systems, it is able to invalidate and/or update data in the local processor's cache as it arrives from the network. Again, FLASH uses a processor core in the NI to help manage all of the protocol processing.

The FLASH architecture utilizes its custom built “MAGIC chip,” which sits between the CPU, memory, the network, and the I/O controller. The ASIC acts as an integrated memory controller and network interface. Integrating the network interface avoids the

overhead of copying network traffic across the system bus. However, as implemented in FLASH, the integration bloats the overhead of the memory controller in servicing CPU memory requests. This added local memory latency exacerbates the already large CPU/memory gap.

The “OS only” programmable FLASH chip make flexible DSM and message passing protocols possible. However, since the protocol processor (PP) offers no particular protection, only the OS can download protocol handlers to MAGIC. By downloading appropriate protocol handlers, it can support several message passing schemes and DSM protocols simultaneously. It can handle OS level active messages, but cannot support user level active messages, as the PP cannot safely execute user code. These limitations imply extra overhead to a user level process, as it must rely on either existing mechanisms or must ask the kernel for support.

This work differs from FLASH in that it uses existing threads on an SMT processor for off loading communications tasks as opposed to requiring a separate communication controller. It also provides more direct access to the network hardware, and provides for message arrival notification to be delivered directly to the user process. The work also looks at the scalable multiprocessor problem from a message-passing standpoint, rather than a shared-memory perspective. If a shared-memory model is desired, that effect could be achieved through software or additional hardware mechanisms.

Wisconsin's Typhoon [97] machine provides a network interface which is placed on the system bus. Typhoon provides a “stock” integer core in the network interface, and allows a user process to program that processor. The dedicated network CPU is intended to handle protocol processing and off load work from the main processor. Since the main CPU

is a traditional processor, Wisconsin designers elected to context switch the network processor along with the regular CPU. Since the current user process has full access to the raw network, the entire cluster must also be gang scheduled to provide protection and ensure message arrival. Gang scheduling can provide extra performance benefits in many cases, since the receive process will in general be active when a message arrives. However, requiring that the machine be gang scheduled adds potentially undesirable restrictions to the conventional computing model.

This work differs from Typhoon in that the NI does not contain a dedicated processor. Instead, contexts within the SMT processor perform message sends and receives. It also does not introduce restrictions that would require gang scheduling.

Princeton's SHRIMP multicomputer [12,13,14,37,38,43] provides a mechanism for mapping memory pages of a local sender process's virtual memory space to a remote receiver process's virtual memory space. Writes to the local memory are forwarded to the corresponding remote memory location as well. Though the map call used in SHRIMP is unidirectional, complementary mappings may be used to setup bidirectional communication. The remote node may be updated either automatically or via explicit "update" commands. In this way, SHRIMP can "separate data movement from destination specification," as the destination is specified when the buffer is mapped. The data is specified by writing to the send buffer (a page in memory). That page may be marked as either an automatic or as a deliberate update. If it is marked as an automatic update, then the data is sent automatically when the data is written in the local processor. If it is marked as a deliberate update, then a send call must be executed to direct the hardware to send the message. Since this

mechanism is supported with page mapping tricks, a minimum send (and hence receive) buffer must be a page.

SHRIMP has no complicated custom protocol processor and no processor core on the NI. The simple design allows for efficient access to the interface, but gives no flexibility to support custom protocols. This work differs from SHRIMP in that is more of a traditional message-passing machine. Upon receives, the local processor is notified with little overhead, allowing the user application to immediately catch and process an incoming message. This provides a bit more flexibility than the SHRIMP design.

9.5 Tighter NIC Integration

In addition to the tight integration of the NIC as a part the M-machine and Avalanche architectures cited above, Nate Binkert explored the benefit of integrating the NIC on the CPU die in his SINIC architecture [10]. Unlike many of the message-passing architectures above, the SINIC architecture focuses on the general IO problem of accessing devices on a logically distant noncoherent IO bus. These key tighter integration principles are also found in several of the message passing and PGAS architectures cited above and are also found in the ULN architecture. The same conclusions apply. The overheads required to interact with a NIC that is on a distant noncoherent legacy IO bus are key to reducing network overheads and thus reducing end-to-end latency. This work differs from Binkert's work in that the ULN architecture includes components of efficient interaction with the network interface related to message arrival notification and in that it couples the efficient NIC interaction with an efficient protocol and system architecture that optimizes ULN as a dedicated message passing architecture.

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

ULN considers message-passing from a holistic point of view targeting messaging overheads and latencies at all levels to provide an architecture that is efficient for fine-grained message-passing. The ULN architecture combines mechanisms for low-overhead send and receive operations, an efficient user-level notification mechanism, efficient inter-thread synchronization, and an SMT processor that can help tolerate message latencies and keep the network busy. The results presented in this work show that this hardware significantly improve the performance of fine-grained message-passing. Indeed, the ULN architecture can be thought of as providing efficient hardware support for active messages.

ULN provides an estimated speedup of 1.51 over an OS-based NIC on DIRT, a fine-grained message-passing code, even assuming a well-optimized OS. A speedup in the range of 1.05-1.32 comes from the inclusion of the user-level notification mechanism, the namesake of the archtecture.

The analysis of DIRT is not only useful in the evaluation of the ULN architecture, it also serves as an example for how other applications can be profiled using TAU or similar tools for evaluation and modeling on ULN or other large-scale architectures. The key elements of that analysis are in understanding the behavior of the node in terms of local computation, synchronization, and communication components. By measuring these elements

and understanding the flow of the code a model can be created similar to the one made for DIRT on the ULN architecture.

The analysis and modeling of DIRT also lead to key insights for how DIRT may be modified to improve performance. Two such what-if scenarios were presented with the associated benefits. Optimizing DIRT for a finer-grained messaging on ULN results in an additional speedup of 1.22 over the original DIRT code. Returning block miss requests directly to the requesting `Renderer` thread instead of to the `communicator` thread may result in an additional speedup of 1.04. The resulting code is factor of 1.9 faster after all optimizations are applied than the original DIRT on a projected NIC with a traditional OS-based interface.

In addition to providing a model for DIRT on the ULN architecture, Chapter 7 demonstrates a Markov model for the expected number of concurrent homogeneous and heterogeneous threads running simultaneously on an SMT processor. This mode can be used to analyze other message-passing codes on the ULN architecture. It can also be used to help model SMT performance on nonmessage-passing applications.

Chapter 7 also includes an extreme-value-theory-driven model for estimating load-imbalance as a function of problem scaling. This model can be used as a model for understanding load-imbalance as a function of scale independent of the ULN architecture. Such a model is particularly useful as we scale into the many thousands or even millions of threads in the Exascale era.

ULN bases the processor model on an SMT processor. Other concurrent, but non-simultaneous multithreaded processors exist. Examples include the Oracle UltraSPARC T1, UltraSPARC T2, UltraSPARC T2 Plus, SPARC T3, SPARC T4, and SPARC T5 pro-

cessors [65] as well as the Cray MTA, XMT, and XMT2 processors [64]. Nonsimultaneous fine-grained multithreading still provides the benefit of managing multiple threads to keep the memory system and network busy, while avoiding some of the complexities associated with SMT processors. It is not immediately clear how such a processor would compare to an SMT processor for these codes. If the performance impact is small, the simplicity could be a big win. An evaluation of other fine-grained concurrent multithreaded processors in the ULN architecture could be evaluated in future work.

The ULN architecture provides a mechanism for user-level interrupts through the use of multiple threads by allowing threads to park awaiting a notification event. Other user-level notification mechanisms would be possible for a single-threaded processor. One such mechanism would be to provide hardware support for a Unix-like signal handler. The hardware could interrupt the currently running thread and switch to a handler on a notification event. To make this possible, the code would need to register an alternate signal-handler stack to the hardware and the hardware would need to be capable of at least automatically saving and restoring a minimal amount on state on a notification.

The notification mechanism in the ULN architecture has the benefit that a wake-up event is a hint. That is, just because a wake-up occurs does not guarantee a message has arrived. It is up to the user-code to check for message arrival upon wake-up. This allows for simplification of the wake-up mechanisms. If the process is not currently running on ULN when a notification arrives, the hardware can safely drop the notification silently. The OS can always trigger a notification when a thread is rescheduled on the processor so that the thread can check for missed notifications, if any. A user-level interrupt mechanism would likely want to retain this benefit by making the trap to the signal handler a hint that

a message may have arrived. In this way, the signal handler semantics would be slightly different than a standard Unix signal handler in that the handler may wake spuriously. Future work could consider such mechanisms for a nonmultithreaded processor.

Investigating user-level interrupts combined with multiple cores as a mechanism for increasing local thread count for tolerating network latencies should be considered. As feature sizes continue to shrink and power becomes a limiting factor, having cores that are underutilized, and are therefore not consuming maximum power is less of a concern than it has been in the past. However, having messages come in to a unified SMT core makes the communication of synchronization events and data values much less expensive, both from a performance and energy perspective. Future work could evaluate the trade-off of a more complex SMT core with cheap interthread communication versus a less complex multicore approach that has higher interthread communication overheads.

Future work should also expand the set of codes analyzed on the ULN architecture. In particular, the MPI library would be a good candidate for optimization on ULN. MPI messages require processing on arrival. Control messages are used to coordinate transfer of large messages and user messages require matching with message tags to determine now where to place the data and what further actions to perform. In this way, an MPI message behaves like an active message.

Finally, there is a lot of interest in mechanisms that increase productivity by, among other things, simplifying the job of programming. This work shows how ULN improves parallel performance, particularly on codes with irregular or unpredictable message patterns. It is the author's belief that providing architectures that efficiently support irregular communication patterns will significantly increase parallel programming productivity.

Providing efficient mechanisms for irregular and fine-grained messaging allows programmers the flexibility to support programming models where remote interaction can be handled in a code at or near the place that it is needed. This helps reduce the amount of programmer effort required to gather communications together into bundles before sending messages. It also saves the programmer the effort of rewriting or even of trying to come up with entirely new algorithms that minimize communication frequency. It would be interesting to study improvements in programming productivity on this new architecture.

REFERENCES

[1] D. A. Abramson, “Computer hardware to support capability based addressing in a large virtual memory,” Ph.D. dissertation, Dept. Comp. Sci., Monash Univ., Melbourne, Australia, 1982.

[2] D. Abts, A. Bataineh, S. Scott, and G. Faanes, “The Cray BlackWidow: A highly scalable vector multiprocessor,” *Proc. ACM/IEEE Conf. Supercomputing*, 2007.

[3] D. Abts, S. Scott, and D. J. Lilja, “So many states, so little time: Verifying memory coherence in the Cray X1,” *Proc. Int. Parallel and Distributed Processing Symp.*, 2003.

[4] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz, “APRIL: A processor architecture for multiprocessing,” *Proc. 17th Annu. Int. Symp. Computer Architecture*, pp. 104-114, 1990.

[5] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, “The MIT Alewife machine: Architecture and performance,” *Proc. 22nd Annu. Int. Symp. Computer Architecture*, pp. 2-13, 1995.

[6] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, and D. Nussbaum, “The MIT Alewife machine: A large-scale distributed-memory multiprocessor,” Massachusetts Institute of Technology, Cambridge, MA, Tech. Memo. LCS-454, 1991.

[7] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, and G. D’Souza, M. Parkin, “Sparcle: An evolutionary processor design for large-scale multiprocessors,” *IEEE Micro*, vol 13, no. 3 pp. 48-61, 1993.

[8] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, “LogGP: Incorporating long messages into the LogP model – one step closer towards a realistic model for parallel computation,” *Proc. 7th Annu. ACM Symp. Parallel Algorithms and Architectures*, pp. 95–105, 1995.

[9] R. Alverson, D. Roweth, and L. Kaplan, “The Gemini system interconnect,” *IEEE 18th Annu. Symp. High Performance Interconnects*, pp. 83 - 87, 2010.

[10] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt, "Integrated network interfaces for high-bandwidth TCP/IP," *Proc. 12th Int. Conf Architectural Support for Programming Languages and Operating Systems*, 2006.

[11] M. Birnbaum and H. Sachs, "How VSIA answers the SoC dilemma," *IEEE Computer*, vol. 32, no. 6, pp. 42-50, 1999.

[12] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, J. Sandberg, "Virtual memory mapped network interface for the SHRIMP multicomputer," *Proc. 21st Annu. Int. Symp. Computer Architecture*, pp. 142-153, 1994.

[13] M. A. Blumrich, C. Dubnicki, E. W. Felten, and K. Li, "Protected, user-level DMA for the SHRIMP network interface," *Proc. 2nd Int. Symp. High-Performance Computer Architecture*, 1996.

[14] M. A. Blumrich, E. W. Felten, K. Li, and M. R. Malena, "Two virtual memory mapped network interface designs," *IEEE 2nd Annu. Symp. High Performance Interconnects*, pp. 134-142, 1994.

[15] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *Ann. of Mathematical Statistics*, vol. 29, no. 2. pp. 610–611, 1958

[16] M. Boosten, R. W. Dobinson, B. Martin, and P. D. V. van der Stok. "A PCI based network interface controller for IEEE 1355 DS links," *Proc. WoTOG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications*, 1998.

[17] B. D. Boschma, D. M. Burns, R. Chin, N. S. Fiduccia, C. Hu, M. J. Reed, T. I. Rueth, F. X. Schumacher, and V. Shen, "A 30 MIPS VLSI CPU," *ISSCC Digest of Technical Papers*, pp. 82–83, 1989.

[18] R. Brightwell, K. T. Predretti, and K. D. Underwood, "SeaStar interconnect: Balanced bandwidth for scalable performance," *IEEE Micro*, vol. 26, no. 3, pp. 41-57, 2006

[19] D. Burger, "Billion-transistor architectures," *IEEE Computer*, vol. 30, no. 9, pp 46-48, 1997.

[20] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes, "Hamlyn: a high-performance network interface with sender-based memory management," Hewlett-Packard, Palo Alto, CA, Tech. Report HPL-95-86, 1995.

[21] K. K. Chan, C. C. Hay, J. R. Keller, G. P. Kurpanek, F. X. Schumacher, and J. Zheng, "Design of the HP PA 7200 CPU," *Hewlett-Packard Journal*, vol. 47, no. 1, pp 25-33, 1996.

[22] S. Chandra, J. R. Larus, and A. Rogers, "Where is time spent in message-passing and shared-memory programs?" *Proc. 6th Int. Conf Architectural Support for Programming Languages and Operating Systems*, pp. 61-73, 1994.

[23] N. Chatterjee, S. H. Pugsley, J. Spjut, and R. Balasubramonian, "Optimizing a multicore processor for message-passing workloads," *Proc. Workshop Unique Chips and Systems*, 2009.

[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general purpose applications on graphics processors using CUDA," *J. Parallel and Distributed Computing*, 2008.

[25] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, 1953.

[26] N. E. Cotter, "Extreme value theory for normal distribution," Private communication, July 2010.

[27] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a realistic model of parallel computation," *Proc. 4th Symp. Principles and Practice of Parallel Programming*, pp. 1–12, 1993.

[28] W. J. Dally and B. Towles, *Principles and practices of interconnection networks*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2004.

[29] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, D. S. Wills, "Architecture of a message-driven processor," *Proc. 14th Int. Symp. Computer Architecture*, pp. 189-196, 1987.

[30] W. J. Dally, J. A. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Fyler, "The message-driven processor: A multicomputer processing node with efficient mechanisms," *IEEE Micro*, vol. 12, no. 2, pp. 23-39, 1992.

[31] W. J. Dally, A Chang, A. Chien, Stuart Fiske, W. Horwat, J. Keen, R. Lethin, M. Noakes, P. Nuth, E. Spertus, D. Wallach, and D. S. Wills, "Retrospective: The J-Machine," *25 Years Int. Symp. Computer Architecture - Selected Papers*, pp. 54-58, 1998.

[32] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley, "Afterburner," *IEEE Network Mag.*, vol. 7, no. 4, pp. 36-43, 1993.

[33] A. L. Davis, M. Swanson, and M. Parker, "Efficient communication mechanisms for cluster based parallel computing," *Communication, Architecture, and Applications for Network-Based Parallel Computing*, pp. 1-15, 1997.

[34] B. Davis, B. Jacob, and T. Mudge, "The new DRAM interfaces: SDRAM, RDRAM and variants," *High Performance Computing*. Springer Berlin/Heidelberg, 2000.

[35] D. E. DeMarle, "Distributed interactive ray tracing for large volume visualization," M.S. thesis, Univ of Utah, Salt Lake City, UT, 2003

[36] D. E. DeMarle, S. G. Parker, M. Hartner, C. Gribble, and C. D. Hansen, "Distributed interactive ray tracing for large volume visualization," Proc. *IEEE Symp. Parallel Visualization and Graphics*, pp. 87-94, 2003.

[37] C. Dubnicki, L. Iftode, E. W. Felten, and K. Li, "Software support for virtual memory-mapped communication," Proc. *10th Int. Parallel Processing Symp.*, 1996.

[38] C. Dubnicki, A. Bilas, and K. Li, "Design and implementation of virtual memory-mapped communication on Myrinet," *Proc. 11th Int. Parallel Processing Symp.*, 1997.

[39] T. H. Dunigan, Jr., J. S. Vetter, J. B. White III, and P. H. Worley, "Performance evaluation of the Cray X1 distributed shared-memory architecture," *IEEE Micro*, vol. 25, no. 1, pp 30-40, 2005

[40] Susan J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: A platform for next-generation processors," *IEEE Micro*, vol. 17, no. 5, pp. 12-19, 1997.

[41] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A user-level network interface for parallel and distributed computing," *Proc. 15th ACM Symp. Operating Systems Principles*, pp. 40-53, 1995.

[42] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A mechanism for integrated communication and computation," *Proc. 19th Int. Symp. Computer Architecture*, pp. 256-266, 1992.

[43] E. W. Felten, R. D. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li, "Early experience with message-passing on the SHRIMP multicomputer," *Proc. 23rd Annu. Int. Symp. Computer Architecture*, pp. 296-307, 1996.

[44] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, "The M-Machine multicomputer," *Proc. 28th Annu. Int. Symp. Microarchitecture*, pp. 146-156, 1995.

[45] A. Gallatin, J. Chase, and K. Yocum, "Trapeze/IP: TCP/IP at near-gigabit speeds," *Proc. USENIX Annu. Tech. Conf.*, pg. 34, 1999.

[46] J. Gu, S. Lumetta, R. Kumar, and Y. Sun, "MOPED: Orchestrating interprocess message data on CMPs," *17th IEEE Int. Symp. High Performance Computer Architecture*, pp. 111-120, 2011.

[47] E. Hecht. *Optics*, 4th ed. Boston, MA: Addison-Wesley Publishing Company, 2001.

[48] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta, "Integration of message passing and shared memory in the Stanford FLASH multiprocessor," *Proc. 6th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 38-50, 1994.

[49] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The performance impact of flexibility in the Stanford FLASH multiprocessor," *Proc. 6th Int. Conf Architectural Support for Programming Languages and Operating Systems*, pp. 274-285, 1994.

[50] D. S. Henry and C. F. Joerg, "A tightly-coupled processor-network interface," *Proc. 5th Int. Conf Architectural Support for Programming Languages and Operating Systems*, pp. 111-122, 1992.

[51] R. W. Hockney, "The communication challenge for MPP: Intel Paragon and Meiko CS-2," *J. Parallel Computing*, vol. 20, no. 3, pp. 389–398, 1994.

[52] R. W. Hockney, "Performance parameters and benchmarking of supercomputers," *J. Parallel Computing* vol. 17, no. 10, 1111-1130, 1991.

[53] R. W. Hockney, "Synchronization and communication overheads on the LCAP multiple FPS-164 computer system," *J. Parallel Computing*, vol. 9, pp. 279-290, 1988.

[54] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy, "The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors," Stanford Univ., Stanford, CA, Tech. Report CSL-TR-95-660, 1995.

[55] B. Jacob, S. W. Ng, and D. T. Wang, with contributions by S. Rodriguez, *Memory systems: Cache, DRAM, disk.* San Francisco, CA: Morgan Kaufmann Publishers, 2007.

[56] R. Jain, *The art of computer systems performance analysis*. New York, NY: John Wiley & Sons, Inc. 1991. p 539.

[57] R. Kalla, B. Sinharoy, and J. Tendler, "IBM POWER5 chip: A dual-core multithreaded processor," *IEEE Micro*, vol. 24, no. 2, pp. 40-47, 2004.

[58] V. Karamcheti and A. A. Chien, “A comparison of architectural support for messaging in the TMC CM-5 and the Cray T3D,” *Proc. 22nd Int. Symp. Computer Architecture*, pp. 298-307, 1995.

[59] V. Karamcheti and A. A. Chien, “Software overhead in messaging layers: Where does the time go?” *Proc. 6th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 51-60, 1994.

[60] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee, “Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor,” *Proc. 25th Int. Symp. Computer Architecture*, pp. 306-317, 1998.

[61] T. Kielmann, H. Bal, and K. Verstoep, “Fast measurement of LogP parameters for message passing platforms,” J. D. P. Rolim, editor, *IPDPS Workshops, Lecture Notes in Computer Science*, vol. 1800, Springer-Verlag, pp. 1176–1183, 2000

[62] J. Kim, W. J. Dally, D. Abts, “Flattened butterfly: A cost-efficient topology for high-radix networks,” *Proc. 34th Ann. Int. Symp. Computer Architecture*, pp. 126-137, 2007.

[63] J. Kim, W. J. Dally, S. Scott, and D. Abts, “Technology-driven, highly-scalable dragonfly topology,” *Proc 35th Annu. Int. Symp. Computer Architecture*, pp. 77-88, 2008.

[64] P. Konecny, “Introducing the Cray XMT,” *Proc. Cray User Group Meeting*, 2007.

[65] P. Kongetira, K. Aingaran and K. Olukotun, “Niagara: A 32-way multithreaded SPARC processor,” *IEEE Micro*, vol. 25, no.2, pp. 21-29, 2005.

[66] D. Koufaty and D. T. Marr, “HyperThreading technology in the NetBurst microarchitecture,” *IEEE Micro*, vol. 23, no. 2, 2003.

[67] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, “The Stanford FLASH multiprocessor,” *Proc. 21st Int. Symp. Computer Architecture*, pp. 302-313, 1994.

[68] K. Langendoen, J. Romein, R. Bhoedjang, H. Bal, “Integrating polling, interrupts, and thread management,” *Proc. 6th Symp. Frontiers of.Massively Parallel Computation*, pp.13-22, 1996.

[69] J. Laudon and D. Lenoski, “System overview of the Origin 200/2000 product line,” *Proc. 42nd IEEE Int. Computer Conf.*, pp 150-156, 1997.

[70] M. Lauria, S. Pakin, and A. Chien, “Efficient layering for high speed communication: the MPI over Fast Messages (FM) Experience,” *Cluster Computing*, vol 2, no. 2, pp 107-116, 1999.

[71] W. S. Lee, W. J. Dally, S. W. Keckler, N. P. Carter, and A. Chang, "Efficient, protected message interface in the MIT M-Machine," *IEEE Computer*, vol. 31, no. 11, pp. 69-75, 1998.

[72] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Trans. Computers*, vol. 34, no. 10, pp. 892-901, 1985.

[73] D.l Lenoski, J. Laudon, K. Gharachorloo, W.-d. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford DASH multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp 63-79, 1992.

[74] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp 39-55, 2008.

[75] J. L. Lo, S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen, "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading," *ACM Transactions Computer Systems*, vol. 15, no. 3, pp. 322-354, 1997.

[76] A. M. Mainwaring and D. E. Culler, "Design challenges of virtual networks: Fast, general-purpose communication," *Proc. 7th ACM SIGPLAN Symp. Principles and Practices of Parallel Programming*, pp. 119-130, 1999.

[77] O. Maquelin, G. R. Gao, H. H. J. Humy, K. B. Theobald, and X. Tian, "Polling watchdog: Combining polling and interrupts for efficient message handling," *Proc. 23rd Annu. Int. Symp. Computer Architecture*, pp. 179 - 188, 1996.

[78] S. Margerm, "Reconfigurable computing in real-world applications," *FPGA and Structured ASIC Journal*, Feb. 2006.

[79] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson, "Effects of communication latency, overhead, and bandwidth in a cluster architecture," *Proc. 24th Int. Symp. Computer Architecture*, pp. 85-97, 1997.

[80] D. Matzke, "Will physical scalability sabotage performance gains?" *IEEE Computer*, vol. 30, no. 9, pp. 37-39, 1997.

[81] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," *Proc. USENIX Technical Conference*, pp. 279-294, 1996.

[82] D. Mizell and K. Maschhoff, "Early experiences with large-scale Cray XMT systems," *IEEE Int. Symp. Parallel Distributed Processing*, pp. 1-9, 2009.

[83] G. Moore, "Cramming more components onto integrated circuits," *Electronics Mag.*, vol. 38, no. 8, pp. 82-85, 1965.

[84] S. S. Mukherjee and M. D. Hill, "The impact of data transfer and buffering alternatives on network interface design," *Proc. 4th Annu. Int. Symp. High-Performance Computer Architecture*, pp. 207-218, 1998.

[85] S. S. Mukherjee and M. D. Hill, "A survey of user-level network interfaces for system area networks," Univ. Wisconsin-Madison, Madison, WI, Tech. Report CS-1340, 1997.

[86] S. S. Mukherjee and M. D. Hill, "Making network interfaces less peripheral," *IEEE Computer*, vol. 31, no. 10, pp 70-76, 1998.

[87] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The J-Machine multicomputer: An architectural evaluation," *Proc. 20th Annu. Int. Symp. Computer Architecture*, pp. 224-235, 1993.

[88] S. Pakin, V. Karamcheti, and A. A. Chien, "Fast Messages (FM): Efficient, portable communication for workstation clusters and massively-parallel processors," *IEEE Concurrency*, vol. 5, no. 2, pp. 60-73, 1997.

[89] S. Pakin, M. Lauria, A. Chien, "High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet," *Proc. ACM/IEEE Conf. Supercomputing*, 1995.

[90] A. Papoulis and S. U. Pillai, *Probability, random variables and stochastic processes.* New York, NY: McGraw-Hill, 2002.

[91] S. Parker, W. Martin, P.-P. Sloan, P. Shirley, B. Smits, and C. Hansen, "Interactive ray tracing," *Proc. Symp. Interactive 3D Graphics*, pp. 119-126, 1999.

[92] S. Parker, M. Parker, Y. Livnat, P.P. Sloan, C.D. Hansen, and P. Shirley. "Interactive ray tracing for volume visualization," *IEEE Trans. Visualization and Computer Graphics*, vol. 5, no. 3, pp. 238-250, 1999.

[93] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. "Interactive ray tracing for isosurface extraction," *Proc. Conf. Visualization*, pp. 233-238, 1998.

[94] V. S. Pai, P. Ranganathan, and S. V. Adve, "RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors," *Proc. 3rd Workshop Computer Architecture Education*, 1997.

[95] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. Fagg, E. Gabriel, and J. Dongarra, "Performance analysis of MPI collective operations," *Proc. 4th Int. Workshop Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, 2005.

[96] N. Rafique, W.-T. Lim, and M. Thottethodi, "Effective management of DRAM bandwidth in multicore processors," *Proc. 16th Int. Conf Parallel Architecture and Compilation Techniques*, pp. 245-258, 2007.

[97] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-level shared memory," *Proc. 21st Annu. Int. Symp. Computer Architecture*, pp. 325-336, 1994.

[98] L. Schaelicke, "Architectural Support for User-Level I/O," Ph.D. dissertation, Univ. of Utah, Salt Lake City, UT, 2001.

[99] L. Schaelicke, A. Davis, and S. A. McKee, "Profiling I/O interrupts in modern architectures," *Proc. 8th Int. Symp. Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pg. 115, 1999.

[100] S. L. Scott, "Synchronization and communication in the T3E multiprocessor," *Proc. 7th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 26-36, 1996.

[101] S. Scott, D. Abts, J. Kim, and W. J. Dally, "The BlackWidow high-radix Clos network," *Proc. 33rd Annu. Int. Symp. Computer Architecture*, pp. 16-28, 2006.

[102] S. Shende and A. D. Malony, "The TAU parallel performance system," *Int. J. High Performance Computing Applications*, vol. 20, no. 2, pp. 287-311, 2006.

[103] E. Spertus, S. C. Goldstein, K. E. Schauser, T. von Eicken, D. E. Culler, and W. J. Dally, "Evaluation of mechanisms for fine-grained parallel programs in the J-Machine and the CM-5," *Proc. 20th Int. Symp. Computer Architecture*, pp. 302-313, 1993.

[104] D. Strenski, "FPGA floating point performance," *HPCWire*, Jan 12, 2007.

[105] E. Suhir, *Applied probability for engineers and scientists*. New York, NY: McGraw-Hill. 1997.

[106] M. Swanson, R. Kuramkote, L. B. Stoller, and T. Tateyama, "Message passing support in the Avalanche Widget," Univ. of Utah, Salt Lake City, UT, Tech. Report UUCS-96-002, 1996.

[107] M. R. Swanson and L. B. Stoller, "Direct deposit: A basic user-level protocol for carpet clusters," Univ. of Utah, Salt Lake City, UT, Tech. Report UUCS-95-003, 1995.

[108] C. A. Thekkath and H. M. Levy, "Hardware and software support for efficient exception handling," *Proc. 6th Int. Conf Architectural Support for Programming Languages and Operating Systems*, pp. 110-119, 1994.

[109] R. Thekkath and S. J. Eggers, "The effectiveness of multiple hardware contexts," *Proc. 6th Int. Conf Architectural Support for Programming Languages and Operating Systems*, pp. 328-337, 1994.

[110] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy, "Supporting fine-grained synchronization on a simultaneous multithreading processor," *Proc. 5th Int. Symp. High Performance Computer Architecture*, pp. 54-58, 1999.

[111] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," *Proc. 22nd Annu. Int. Symp. Computer Architecture*, pp. 392-403, 1995.

[112] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16 no. 2, pp. 28-40, 1996.

[113] D. L. Weaver and T. Germond, *The SPARC architecture manual*, version 9. SPARC International, Inc., Upper Saddle River, NJ: PTR-Prentice Hall, 1994.

[114] M. Welsh, A. Basu, and T. von Eicken, "ATM and Fast Ethernet network interfaces for user-level communication," *Proc. 3rd Int. Symp. High Performance Computer Architecture*, pp. 332-342, 1997.

[115] M. Welsh, A. Basu, and T. von Eicken, "Incorporating memory management into user-level network interfaces," *IEEE 5th Annu. Symp. High Performance Interconnects*, 1997.

[116] Advanced Micro Devices, Inc., *AMD home page*. [Online]. Available: http://www.amd.com

[117] *Alpha architecture handbook*. Houston, TX: Compaq Computer Corporation, 1998.

[118] ClearSpeed Technoogy, Ltd., *CSX processor architecture whitepaper*. [Online]. Available: http://www.clearspeed.com/docs/resources/ClearSpeed_CSX_White_Paper.pdf

[119] *FBDIMM architecture and protocol*, JEDEC Standard JESD206, 2007.

[120] *FireWire specification*, IEEE Standard 1394, 1995.

[121] *HyperTransport I/O link specification*, HyperTransport Consortium Standard 3.10, 2008.

[122] *The InfiniBand architecture specification*, InfiniBand Standard 1.2, 2004.

[123] Intel Corp., *An introduction to the Intel® QuickPath Interconnect*, 2009.

[124] Intel Corp., *Intel 7500 Scalable Memory Buffer datasheet*. [Online]. Available: http://www.intel.com/content/www/us/en/chipsets/7500-7510-7512-scalable-memory-buffer-datasheet.html

[125] Intel Corp., *Intel home page*. [Online]. Available: http://www.intel.com

[126] International Business Machines Corp, *IBM home page*. [Online]. Available: http://www.ibm.com

[127] *International technology roadmap for semiconductors*. Semiconductor Industry Association, 1998.

[128] *More AMD G3MX details emerge*. [Online]. Available: http://hotharware.com/News/More_AMD_ G3MX_Details_Emerge/

[129] Motorola Semiconductor Product Sector, *RapidIO: An embedded system component network architecture*, 2000.

[130] *MPI: A message-passing interface standard*, Message Passing Interface Forum Standard UT-CS-94-230, 1994.

[131] *PCI Express 3.0 base specification*, PCI-SIG Standard PCI Express 3.0 Base, 2010.

[132] *PCI Local Bus specification revision 3.0*, PCI-SIG Standard Conventional PCI 3.0, 2002.

[133] Tilera Corp., *Tilera home page*. [Online]. Available: http://www.tilera.com

[134] *Top 500 supercomputer sites*. [Online]. Available: http://www.top500.org

[135] *Universal Serial Bus 3.0*, USB Standard 3.0, 2008.

[136] Wolfram Math World, *Gumbel distribution*, Available: http://mathworld.wolfram.com/GumbelDistribution.html