TRAVELING THREADS: A NEW MULTITHREADED EXECUTION MODEL

A Dissertation

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

Doctor of Philosophy

by

Richard Cameron Murphy, M.S., B.S., B.A.

Peter M. Kogge, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
June 2006

TRAVELING THREADS: A NEW MULTITHREADED EXECUTION MODEL

Abstract

by

Richard Cameron Murphy

Computer architecture is plagued by the von Neumann bottleneck. This work introduces and evaluates the *traveling thread* execution model in which threads migrate to the memory resources close to the data they require rather than perform remote memory accesses. This helps address the von Neumann problem by exposing additional concurrency within programs to tolerate long memory latencies, reduces two-way request/response network transactions typical of caching architectures to one way thread migration transactions, and reduces or eliminates cache coherency traffic.

For Sarah, who always inspires me.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Peter Kogge, for his invaluable guidance, support, and contributions to this work. I would like to thank him most of all for allowing me to attempt to answer a "big" question, and for pointing me in the right direction whenever I got lost in the middle of doing so. I am grateful to him every day that this dissertation had the topic that it did.

My wife, Sarah, demonstrated invaluable support, patience, and insight every time my "Eurekas" turned out to be premature.

My parents and grandparents started me down this path long ago, supported me along the way, and will be very happy now that I have finished. Mom, your love of learning is infectious; and Dad, your nagging did not really make this go any faster, but I appreciate it anyway.

This work would be impossible without the support and intellectual guidance of my committee, who should also know that I appreciate the time they took reading this. Thanks to Sharon Hu, Danny Chen, and Kevin Bowyer.

To my friends and colleagues for their input (and sometimes shared insanity): in particular, Arun Rodrigues, for looking at half a million lines of Fortran and saying "I will get back to you before lunch", and for claiming that, in some small way, I vexed him; Jeff Squyres and Brian Barrett for explaining the innumerable ways in which MPI is broken; Shannon Kuntz for reminding me the end was in sight; and Mike Niemier for being Mike Niemier.

Special kudos go to my friend, F. Nicholas Rahaghi for always reminding me that I should treat my advisor like I would Yoda, that getting a Ph.D. in pretty much

any field is roughly the same experience, and that he would mock me mercilously for the next 30 years if he finished first.

This work would not have been possible without the support and help of a number of individuals, particularly in the national labs. I would like to thank the following individuals in particular:

- Keith Underwood deserves special thanks for actually making everything work, and finding the resources necessary to perform many of the computational results presented in this dissertation. I also thoroughly enjoyed and benefited intellectually from the "It's the Latency... no, it's the Bandwidth" debate. However, I still think it's the latency.
- Erik DeBenedictis proved an excellent sounding board for crazy ideas.
- Neil Pundit provided invaluable support, and guidance.
- Steve Plimpton gave LAMMPS, the appropriate input sets for it, and valuable assistance in describing those inputs.
- Sue Goudy guided the acquisition, compilation, execution and selection input sets for CTH. She also helped to track down descriptions of those input sets, and to sort through half a million lines of Fortran when necessary. Most significantly, she took it in stride whenever the results were surprising.
- Mike Heroux and Alan Williams provided invaluable help with Cube3 and Trilinos, as well as innumerable new bits of linear algebra and numerical analysis to examine.
- Ron Brightwell managed to make even MPI look somewhat sane (at times).
- Jeffrey Vetter and John Engle helped with sPPM.

Finally, my profound thanks go to Burton Smith and Thomas Sterling. Burton somehow helped communicate the zen of multithreading to me, told me I was wrong when I was, and showed me his famous penny stacking trick. Thomas reminded me that this was my first big research project, not my last, and used the term "exothermic synergism" when it was called for.

This work was funded in part by Sandia National Laboratories, and in part by the Defense Advanced Research Projects Agency (DARPA) under its Contract No.

NBCH3039003. The PIM architectures and execution models are outgrowths of many previous projects funded by DARPA, JPL, and other agencies.

An early part of this effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under Cooperative Agreement number F30602-98-2-0180. The U.S. Government is authorized to reproduce and distribute for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Rome Laboratory, or the U.S. Government.

CHAPTER 1

INTRODUCTION

The primacy of single processor performance, and the ease of the corresponding programming model, has forced the architects of parallel machines to accept limited scalability of processor and memory hierarchies. The fundamental inability to construct truly scalable hardware and software execution models impacts the construction of high-end systems, particularly those capable of exceeding petaflop performance. Initiatives such as the original IBM BlueGene/C [5, 108] have addressed performance via the use of many processing elements, but are fundamentally limited in scalability and programmability. BlueGene/C, in fact, supports a limited number of applications, primarily protein folding using energy decompositions with highly replicated data, and is not a truly "general purpose" machine. Furthermore, the reliance upon limited application scope to achieve petaflop performance belies the benefit of that performance, particularly considering the investment such machines require.

The Massively Parallel Processor (MPP), typified by machines such as Sandia National Lab's Red Storm, represents the workhorse of domestic supercomputing. MPPs are similar to commodity "clusters", which are constructed of entirely off-the-shelf parts, but are generally constructed with custom interconnection networks. The current generation, providing tens of teraflops of capability, is used to perform critical analysis, including the stewardship of an aging nuclear stockpile, analysis of

complex engineering problems, molecular design and analysis, and, more generally the simulation of physics on a computer. However, even the MPP's most vocal advocates admit that within a machine generation or two (many, especially critics, would argue earlier) the fundamental system architecture of MPPs will be unable to efficiently scale any further. Indeed, given the low utilization of current machines, this inefficiency represents costly overhead for any machine with a reasonable, finite power budget. As we pursue trans-petaflop scale machines, we are simultaneously faced with tremendous engineering challenges, and potential rewards of benefit to humanity: including, the prediction of an earthquake's interaction with the topology of a given area and a feedback to the design of buildings, complex physiological modeling, and applications vital to national security.

Today's large parallel machines come in a spectrum from the MPP consisting of tens of thousands of loosely coupled nodes, providing message-passing parallelism, to Cache Coherent Non-Uniform Memory Access (CC-NUMA) architectures consisting of dozens to thousands of nodes, providing stronger coupling and thread level parallelism, or various combinations in between. Most significantly, the memory wall looms, and while interconnection networks provide more bandwidth, they have done little to address the problem of relative latency. Indeed, the supercomputer community's adage is that providing more bandwidth is merely expensive, whereas reducing latency is next to impossible, especially relative to the cycle time of a processor. In a typical machine that includes both near (on-node) and far (off-node) memory, both types of data are more distant from the processing resources. The smaller machines, which typically provide cache coherency, must also cope with the overhead involved in that coherency traveling over the same relatively slow links. Beyond that, the nature of supercomputing applications is changing. Today's application base is more typified by very sparse, irregular memory accesses (parallel

graph algorithms, the Giga-Updates Per Second, or GUPS benchmark, etc.), which place an even larger burden on the memory system than traditional sparse matrix operations. The problem is exacerbated by the shift from specialized architectures (such as vector machines, which are capable of impressive performance on linear algebra codes) to MPI-programmable clusters (commodity or otherwise).

Furthermore, regardless of the underlying architecture, the programmer is forced to explicitly identify the available parallelism in an application at a coarse-grain level. For supercomputing applications, this is typically done structurally: that is, when performing the simulation of physics on a computer, the programmer decomposes the problem in both space and time, and programs the machine according to that structure. However, attempting to extract more parallelism structurally by increasing the problem's resolution leads to exponential demands on processing and communication resources. For example, doubling the spatial resolution and halving the time step requires an n^4 increase in compute time, and an n^3 increase in memory. Quite simply, larger machines require significantly more opportunities to exploit concurrency than today's MPPs, and that concurrency may be difficult to continue to extract solely via structural decomposition.

This dissertation begins to address critical problems facing the construction of trans-petaflop scale architectures. Specifically, it examines extremely light-weight methods for identifying and exploiting fine-grain concurrency in parallel applications, tolerating the additional latency imposed by the von Neumann bottleneck, and extracting additional concurrency from programs. This is combined with a physical architecture capable of significantly reducing local latencies.

Processing-In-Memory (PIM) [61, 62, 20, 79, 78, 77] exploits tremendous amounts of memory bandwidth available for intra-chip communication and, therefore, circumvents the von Neumann bottleneck by placing logic and memory on the same die.

This allows for the construction of highly distributed systems with a tremendous latency gap between high speed local memory macro accesses and remote accesses. They demonstrate, precisely, the most challenging problem described above. In the short- to medium-term, the toleration of very long latencies through simple execution models is of paramount importance for the implementation of machines of gargantuan scale. It allows for on-chip real estate to be consumed performing useful work (by exploiting concurrency!) rather than maintaining complicated, state rich coherence and latency toleration mechanisms.

Though currently seen most vividly in the form of the von Neumann bottleneck, in the long run, the problem of retrieving data for processing will only become exacerbated. High-end microprocessor implementations have seen pipeline structure dictated by cross-chip clocking requirements. Some non-transistor technologies, such as QCA [71, 9, 70], dictate device architectures dominated by pipelining requirements, which requires latency toleration at all levels throughout the system, not merely the memory hierarchy.

The traveling thread execution model discussed here proposes a novel multithreaded execution model targeted at an extremely large PIM array capable of trans-petaflop performance. In a *multithreaded* machine, the program's execution is divided into multiple streams of instructions (or threads) that can potentially be executed concurrently (as opposed to a single-threaded system that supports only one such stream). A *traveling thread* is any thread in a multiprocessor system that, when referencing remote data, migrates to the node in the system where that data resides rather than performing a remote read and write. And the traveling thread execution model (described in detail in Chapter 4) describes the architectural features (and trade-offs) required for a thread to make that transition. This execution model is explored in terms of its impact on the construction of the array's processor

and memory hierarchy, its performance, and the corresponding programming model. As the number of processors in a system with a shared, globally accessible address space increases, both the latency of a remote access and the overhead associated with maintaining *memory consistency* (across processors) increase tremendously. Unlike prior work which focuses upon performance measured in a single dimension: either (most popularly) its raw latency avoidance performance, or, the available concurrency in the system, this work explores the trade-offs in both dimensions. Furthermore, additional attention is paid to a tightly coupled programming model and its potential to deliver a system capable of high *productivity*. Thus the proposed execution and programming models address both performance and the additional design parameters associated with the multi-generational, highly scalable architecture required of a general purpose machine capable of executing diverse workloads. Both problems are addressed simultaneously by defining and quantitatively analyzing a novel multithreaded execution model that permits individual threads to migrate, ideally to CPUs that are "nearer" to data of interest coupled with a PIM architecture that moves some large section of memory physically closer to the processor. This model is coupled with a very large, non-coherent SMP-like architecture consisting of millions of Processing-In-Memory (PIM) nodes and, potentially, tens of thousands of heavyweight processors which work in concert with the PIMs.

1.1 The Problem

Globally, von Neumann computing is afflicted by latency. The raw processing speed required for logic performance, and the high density required of storage devices lead to largely incompatible design choices. Indeed, this disparity is evident from the early days of electronic computers where high speed logic, in the form of vacuum tubes, was coupled with high density memory, in the form of drums. Fundamentally,

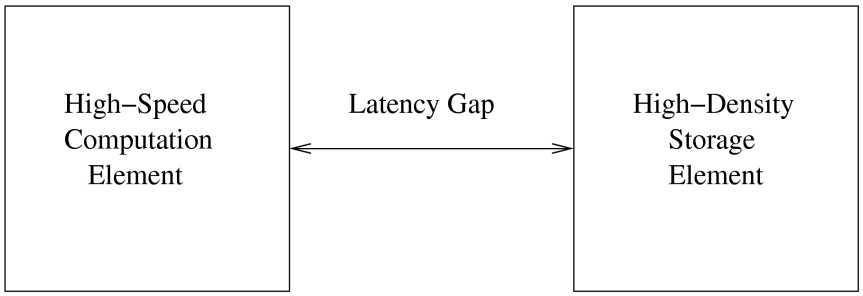


Figure 1.1. The von Neumann Bottleneck

this is no different from the coupling of high speed processors with relatively low-speed DRAM. Figure 1.1 provides a simple depiction of the problem.

Fundamentally, there are only two ways of combating the problem:

1. **Latency Toleration:** which, instead, focuses on the throughput of the system by performing useful work when latency "waiting" would otherwise be required. This work provides for extensive latency toleration by allowing the program to increase concurrency by defining a spectrum of threads.
2. **Latency Avoidance:** which constitutes an attempt to increase performance by dedicating hardware to eliminate some of the latency. In this work, latency is avoided in two ways: first and foremost, round-trip (request/response) latencies are converted to one-way latencies in the form of traveling threads; secondly, those threads are coupled to PIMs, which move very large pieces of the address space physically closer to the processor.

Significant modern computer architecture research focuses on techniques for Latency Avoidance. Caching, for example, avoids latency by placing small amounts of high-speed memory near the processing logic in the hopes of capturing a significant working set. References outside that set, in the form of a cache miss, cannot be avoided. Other modern techniques, such as Simultaneous Multithreading (SMT) [56, 75] and out-of-order execution focus on tolerating high latency operations by overlapping other, useful work during memory accesses.

The problem was, perhaps, best stated by von Neumann himself in 1946, when describing the "memory organ" of a computer[26] and is frequently cited in general

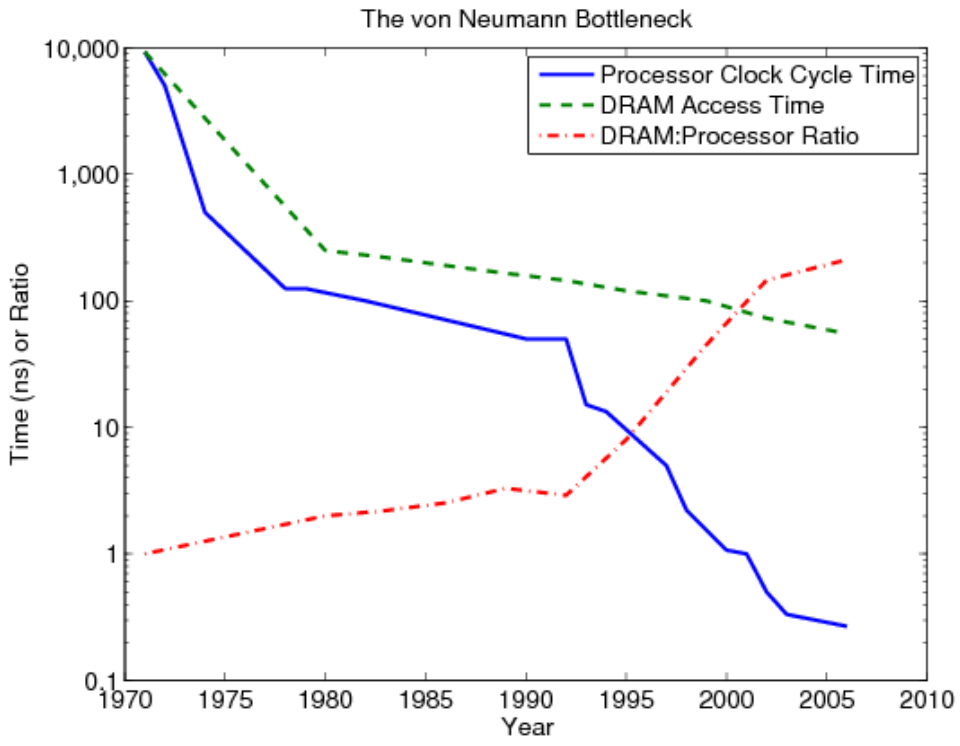


Figure 1.2. The von Neumann Bottleneck

computer architecture texts[87, 51]:

> Ideally one would desire an indefinitely large memory capacity such that any particular... [memory] word... would be immediately available... It does not seem possible physically to achieve such a capacity. We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

The von Neumann bottleneck has been the primary factor in driving computer architecture research over the past six decades. Each advance in architecture has provided a mechanism for latency avoidance, latency toleration, or both (either directly or indirectly).

Figure 1.2 depicts the challenge posed by the von Neumann bottleneck (often called the Memory Wall[107]) for Intel's processor line from the introduction of the

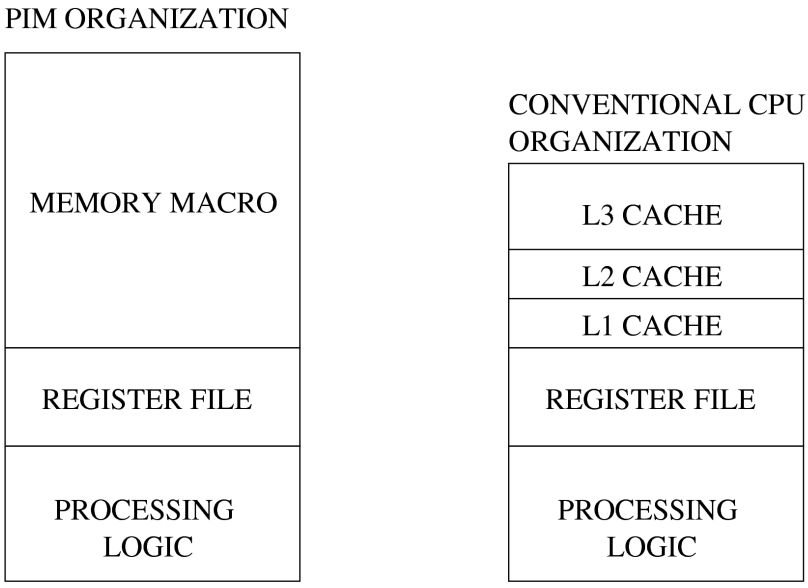


Figure 1.3. PIM Configurations

4004 in 1971 to today. The curve demonstrates a radical increase in the ratio of memory access time to processor cycle time (over 210 fold in less than 4 decades). This is, most fundamentally, the problem facing computer architects, and the problem addressed in this dissertation. The PIM systems discussed in this work have explicit mechanisms for both latency avoidance (in the form of moving the memory physically closer to the processor and exposing more bandwidth), and latency toleration (in the form of multithreaded execution).

Figure 1.3 shows the basic memory configuration of a PIM. Relatively large amounts of on-chip memory (SRAM or DRAM) are available to the PIM directly. Generally speaking, this memory can be used as a cache or a paged space. PIMs can further be interfaced with the rest of the system in any system architecture.

In the case of PIM systems, the memory hierarchy itself becomes more extreme, causing the application of a single latency strategy to be difficult, or impossible. In fact, even the design of "conventional" machines is beginning to reflect a balance

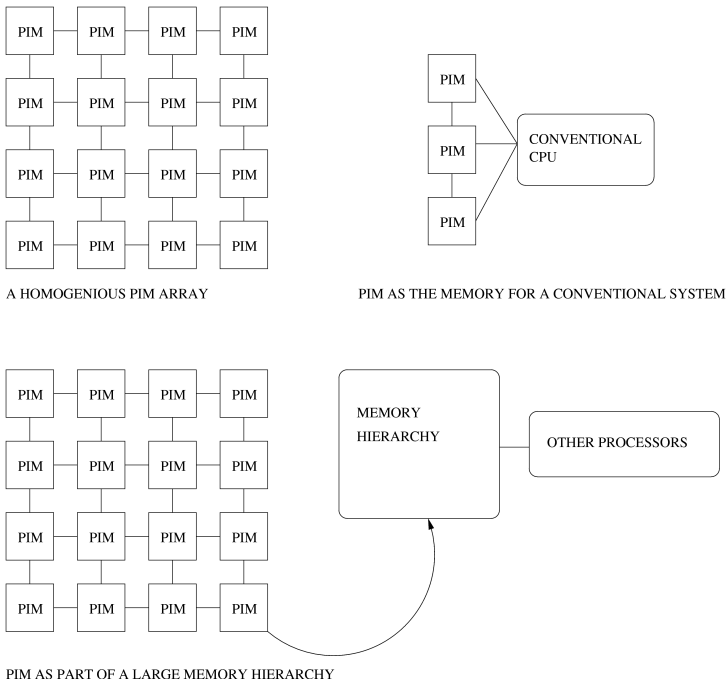


Figure 1.4. PIM Array Configurations

between latency toleration and avoidance. For example, modern processor design could be considered centered upon latency avoidance (simply put, on-chip caches dominate the design). By contrast, the Cray/Tera MTA[8] focuses entirely upon Latency Toleration. The latest designs, such as recent multithreaded incarnations of the PowerPC[19] and Pentium IV show the increased blending of the two strategies.

Consider the possible PIM configurations shown in Figure 1.4. In terms of configuration, the obvious forms are: a singular, homogeneous array of PIMs, an array of PIMs acting as the entire memory system (and supplemental processor) for another CPU, or a heterogeneous array in which PIMs, and other storage and processing elements work cooperatively. While in general, this work concentrates only on the PIM-to-PIM interactions, the general latency problem remains the same for all configurations. Furthermore, the hierarchical nature of the problem becomes extreme as PIM arrays must consist of hundreds of thousands or millions of nodes to achieve

trans-petaflop performance.

The PIMs themselves have relatively high-speed/high-density **local** memory access: that is, the first part of the memory hierarchy has strong characteristics in both density and performance. However, once the useful data in that first level is consumed, the PIM must endure a very long latency operation to obtain new data, regardless of its location: on another PIM, off-chip storage, or, potentially further away.

The traveling thread execution model examines the additional costs of breaking an instruction stream down into threads of varying length, as well as the potential to manage high-latency remote memory reference transactions via migration instead of the typical request/response protocol used in today's shared memory parallel machines.

1.2 Objectives and Results

This dissertation defines and evaluates the *traveling thread* execution model's design space, in which threads move throughout a PIM array in search of their data, rather than the data being brought to a particular thread on demand. This analysis is performed in the context of trans-petaflop PIM arrays, and the applications most likely to be executed on them. Included herein is a canonical implementation of the execution model.

This dissertation seeks to understand:

1. What the memory characteristics of real-world supercomputer applications are compared to the benchmarks typically studied by computer architects;
2. When threads capable of moving to their data are useful: specifically, under what circumstances such a thread can reduce the overall latency in a computation by converting two-way (request/reply) latencies into one way latencies, and how the compiler writer can use this feature of the execution model to boost performance;
3. How much and what type of data needs to accompany a thread as it migrates through the system;

4. How this execution model affects the overall concurrency of the program and the delineation of individual threads;
5. When the compiler may be able to automatically identify and issue very small threads;

The results demonstrate that:

1. Real world supercomputing applications demonstrate radically more complicated memory performance than the most studied computer architecture benchmark suite.
2. Conventional cache architectures would be required to support up to 40 times the number of simultaneous memory transactions than they do today to outperform the traveling thread execution model.
3. Traveling threads can have their state packaged into typical cache-line or smaller sizes.
4. Traveling threads can be used to potentially increase the explicit concurrency in the system by orders of magnitude, with synchronization costs varying with the length of the thread.
5. Traveling threads potentially exist as a replacement for a remote caching memory reference, and that the compiler's primary concern in determining thread length is the cost of sharing data between threads in terms of synchronization.

In contrast to prior work[77], this work focuses on the theoretical potential to extract traveling threads from existing instruction streams. It provides the architect with the ability to quickly estimate the performance of real-world applications once architectural features have been given a cost (e.g., once a method for performing atomic synchronizations is defined with an associated cost, how the overall application performance changes can be estimated). And it provides the compiler writer with a new execution model that can extract additional parallelism from the application explicitly, and in particular what types of traveling threads can be extracted. The results are used in Chapter 8 to describe a theoretical traveling thread machine that selects the architectural features supported by the experiments (from the range described by the execution model in Chapter 4).

1.3 Dissertation Outline

In support of demonstrating the success of the above enumerated objectives, this dissertation evaluates the *traveling thread* execution model. It begins by defining the model itself, and identifying the key trade-offs necessary during the model's implementation. The key metrics necessary to correctly evaluate the proposed model are identified and broken into two categories: first, metrics which evaluate latency avoidance; and, second, metrics which evaluate latency toleration. The trade-offs between the identified metrics are examined in terms of current parallel benchmark suites, using today's codes.

The remainder of this dissertation is organized as follows: Chapter 2 provides a review of prior work and a context for the experiments given in this dissertation. Chapter 3 describes the parallel benchmarks under study, and compares them to the most popular computer architecture benchmarks in terms of memory performance. Chapter 4 formally sets forth the traveling thread execution model. Chapter 5 is an analysis of dataflow graphs, focusing on the potential for concurrency offered in today's instruction streams. Chapter 6 breaks the dataflow graph into multi-instruction threads, and examines the trade-offs of doing so, particularly in terms of synchronization. Chapter 7 analyzes the partitioning of data between different memory nodes in the traveling thread model. And Chapter 8 concludes by using the experimental results from this work to describe a theoretical traveling thread machine (based on the requirements from Chapter 4), and summarizes the conclusions and potential for future work.

CHAPTER 2

THE STATE OF THE ART CONCISELY DEFINED

There is a large body of work devoted to the study of parallel computer architecture, and parallel memory systems in particular. This chapter serves as a general guide to the methodology used to extract program information and traveling thread characteristics (Section 2.1). In addition, this chapter describes the relevant computer architecture advances, such as processing-in-memory architectures (Section 2.2); memory architectures and consistency models (Section 2.4); threads and multithreading (Section 2.5). Programming Models which may map well onto the proposed system (Section 2.6) are described, and finally, Section 2.7 concludes by discussing the architectural relevance of these topics to PIM.

2.1 Analysis and Simulation Methodology

There are multiple levels of program analysis employed in this work. Each of them begins with a real-world *instruction trace* that records each instruction executed by an application, as well as the relevant changes in machine state (e.g., which memory addresses are accessed, which branches are taken, etc.). The instruction trace serves as a record of program execution, and the actions taken by a particular processor. In this work, the traces are used to show the theoretical capability of transforming those streams into a traveling thread execution stream. Critically, this allows for the study of what the architectural trade-offs for a traveling thread are,

what the thread should look like, and what the upper bound in performance should be given today's instruction streams. Methodologically, this allows for a larger exploration of the design space than would be permitted by constructing a small set of traveling thread implementations and benchmarking the performance. Additionally, it allows for the architecture work to specify what traveling threads should look like to the compiler writer, rather than relying on a particular compiler writer's implementation of a thread extraction algorithm. This is particularly important given the challenge of developing a new architectural feature without having first developed the compiler to exploit it.

The instruction traces in this work are extracted via Apple's Amber framework (see Section 2.1.1), which is similar to the Sun Shade suite [100]. The trace information, particularly the extracted memory reference traces and thread dependencies, is propagated to higher-level analysis programs, often by building graph models of the problem under study based on the serial trace, and determining the properties of those models. Individual modeling is discussed with the experimental setup for each chapter that follows. Of critical importance is the trace input data used to drive each experiment in this dissertation.

2.1.1 Amber

One mechanism for benchmarking throughout this work is the Amber instruction trace generator, which is a component of Apple's Computer Hardware Understanding Toolkit[11]. The tool allows PowerPC binaries to be analyzed in detail by providing a simple mechanism for analysts to capture the instruction stream of any executing application. Amber provides information about every instruction executed, and, optionally, the full register state changes that each instruction causes on the PowerPC's machine state.

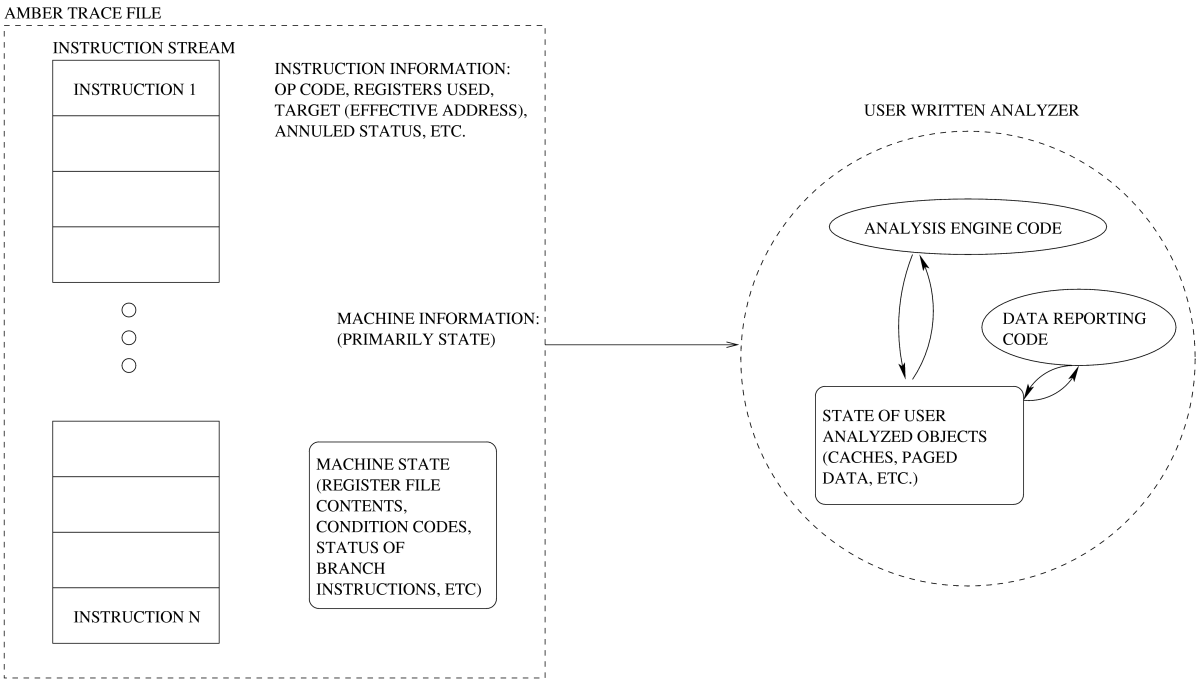


Figure 2.1. Amber Simulations

As Figure 2.1 shows, programs are viewed simply as streams of instructions. Each simulator written for the purposes of this dissertation uses those streams of instructions, combined with information Amber provides about the state of the machine, to perform accounting for whatever is under analysis.

It should be noted that Amber is incapable of tracing calls inside the kernel, and therefore cannot account for system overhead. For the purposes of this work, where no particular PIM runtime system implementation exists, excluding profiling of the runtime system is extremely beneficial, as the focus is on application requirements. In fact, in most supercomputers, the runtime system is designed to be as small and efficient as possible, and provide only the functionality necessary to perform the computation.

2.1.2 Validation

Each individual analysis tool (presented in Chapters 3-8) was validated using artificially generated input traces with known properties (memory access patterns, instruction mixes, possible concurrency, etc.) as well as, in the case of memory system simulations, against actual hardware or well established system simulators, such as SimpleScalar[23]. For example, each experiment involving cache architecture (see Chapter 3) was validated using simple programs that, rather than representing a real application, produce results with known memory properties.

- **Local**: all memory accesses are local to a single node. The trace consists of a program that increments each element of a 1 MB array repeatedly. Each element in the array is incremented once, causing both a load and a store. This is useful in the case of a traveling thread because none of the threads in this trace should ever move.
- **Ping-Pong**: all memory accesses are between two nodes on opposite ends of the address space, and thus always remote. The trace consists of a copy between 1 MB arrays with a starting memory location at the beginning of the address space, and one with a starting memory location on the other side of the address space. This is done by allocating one array on the heap and one on the stack. In the case of a traveling thread validation, it provides for threads which must always move between two nodes.
- **Random**: all memory accesses are random and distributed uniformly throughout the address space. This trace was generated artificially in that it conforms to the Amber trace format but does not actually represent the execution of a program that was run. Instead, it consists of a series of alternating load and store instructions with memory addresses generated by a random number generator. In the case of a traveling thread, this represents threads that are distributed uniformly throughout the memory array.

To ensure that all PowerPC instructions are properly decoded, an artificially generated trace containing every opcode repeated 10 times and randomly ordered is used. Every analysis tool reports an instruction mix at the end of analysis for a given trace, and those mixes are compared against known values, and other simulations to ensure that instruction interpretation is consistent everywhere. Every analysis tool also maintains two sets of instruction mix counters: one generated strictly by

the opcode of each instruction, and one generated by instruction type (e.g., ALU operations, memory references, branches). These are compared for validation across instruction classes (e.g., that the correct number of compute, memory, and branching instructions have been properly interpreted).

In the case of analysis tools providing concurrency measures, three artificial traces were generated:

- **Serial**: performs a series of dependent instructions that do not provide concurrency (e.g., the result from the previous calculation is used in the next calculation). This is done by computing a Fibonacci sequence using two registers in the following (pseudo-code) loop:

```
register int a=0, b=1;

while(1) {
   a = a+b;
   b = b+a;
}
```

- **Concurrent**: performs a series of completely independent loads clustered on a local node, without any non-memory operations between them. This creates a series of threads that require no remote data, and are highly concurrent.
- **Random**: performs a series of instructions random data dependencies, that provides a uniform distribution of threads that require significant remote data.

2.2 Processing-In-Memory (PIM)

Processing-in-Memory (PIM, also known as Intelligent RAM [86], embedded RAM, or merged logic and memory) takes advantage of the development of fabrication technology capable of merging high-density DRAM and high-speed logic on the same die. The physical design process enabling PIMs has moved from two completely separate processes being applied to the same die (one for DRAM and one for logic), which is both technically challenging and costly, to starting with either a base memory process and adding a few fabrication steps for logic, or a base logic process and adding a few fabrication steps for DRAM. The former (logic-in-memory) process

produces denser DRAM and somewhat slower logic, while the latter (memory-in-logic) produces both faster logic and memory, but less dense DRAM (and leakier transistors).

2.2.1 The Architecture of Other Large-scale PIM Efforts

This change in fabrication technology allows for chips to be created that exploit the low latency, high bandwidth interconnection between processor and memory on the same chip. Several proposals for what the technology could achieve have been made. The Intelligent RAM (I-RAM) project at Berkeley placed a general purpose core with vector capabilities along with a memory macro onto a die for embedded applications. Cellular phones, personal digital assistants, and other devices requiring processing power and small amounts of memory could benefit tremendously from this type of system, even if one only considers the potential advantages in power consumption.

Other projects, such as the Galileo group at the University of Wisconsin [22, 25, 24] have seen PIM as having tremendous potential for use in standard workstations. The on-chip memory macro becomes either a part of the memory hierarchy (as either a space for caching data or paging it from DRAM), or when the memory density becomes high enough, the entire memory hierarchy. Both of these views see PIM as a technology which fits very definitely into the framework of contemporary computer architecture, except that increased memory bandwidth is available. This can be seen in that each group proposes placing a fairly large, traditional (and expensive) processor core on the die. Naturally there are some additional capabilities which can be exploited (such as the addition of register level vector instructions which can read the larger words available from memory).

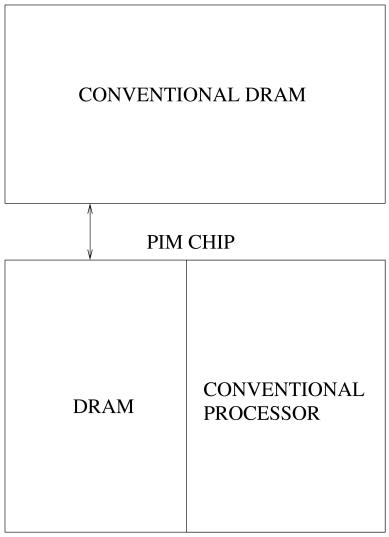


Figure 2.2. Conventional PIM Proposals

Figure 2.2 shows the basic model for both IRAM and Galileo-based PIMs, in which DRAM is merged with a (more or less) conventional superscalar out of order processor, which may or may not include backing DRAM for memory access. How the on-chip memory is organized may differ: as a paged memory space, as a cache, or as the entire memory for a small device (e.g., a cell phone), but, fundamentally no large-scale changes to the processor are proposed. It is an attempt to leverage the low-latency high-speed on-chip memory without exposing much of that memory to the programmer's control.

2.2.2 A PIM's View of Memory

A conventional processor largely views memory as a large, uniform structure (or, at least that is the view that it presents to the programmer). Caches are generally intended to implicitly provide latency avoidance by managing the program's temporal and spatial working set automatically. Chapter 3 discusses the temporal and spatial working sets in great detail, but the *temporal working set* is the set of

memory locations that an application has accessed most recently; and the *spatial working set* is the set of memory locations near locations the program has used that have not themselves been consumed, but will be in the near future. A conventional cache attempts to capture both: the number of blocks in the set provide for unique windows into the address space (or, roughly, temporal reuse), while the block size provides for spatial reuse by reading more data than was requested during any given memory read. Despite the complexity, the programmer is given little or no control over how this is done (what data items should be cached because they may be reused, etc.).

A PIM presents a different view to the programmer: memory is non-uniformly accessed (e.g., local memory is lower latency than remote memory), and because the memory on PIM nodes in this work is part of the overall machine's explicit address space, the programmer can control data placement (which Chapter 7 demonstrates can significantly improve performance). Doing so in a conventional architecture is much more difficult because of the complexities of the cache architecture.

2.2.3 Technology and Performance

This work takes the view that PIM is a technology which enables an entirely new computing model in which potentially millions of nodes can be utilized in tandem for general applications. This line of thought stems from a long history of research, covered in detail in Section 2.3, including the DIVA and HTMT projects.

Figure 2.3 compares the available bandwidth to various forms of memory on a 1 GHz PIM versus a 2 GHz UltraSPARC IV-like RISC processor. The x-axis shows the bytes of reachable memory. In the conventional case, it starts with the register file's available data storage memory, followed by the L1 and L2 caches, and, finally remote or main memory. In the PIM case, it shows the register file's available memory,

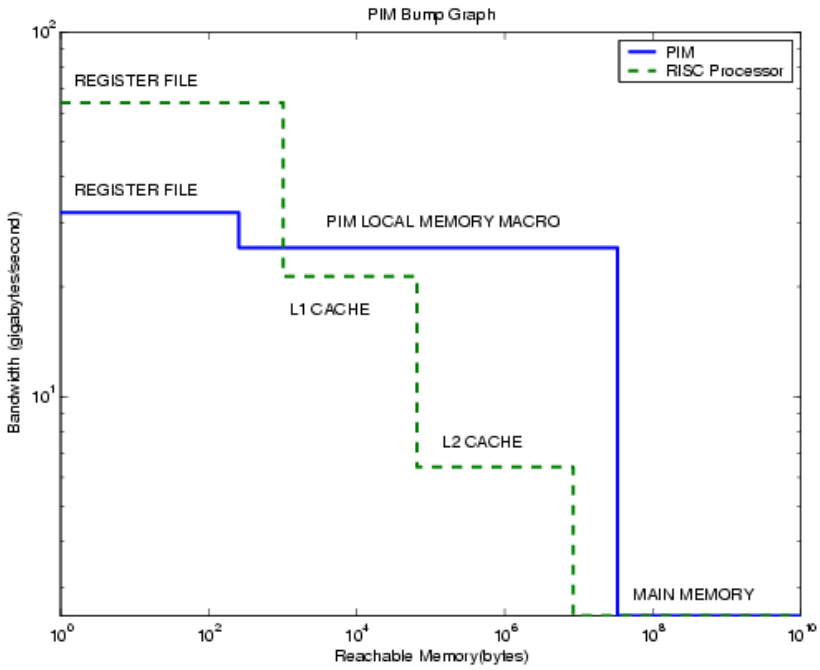


Figure 2.3. PIM Memory Bandwidth

followed by the local PIM memory, and finally external memory. The external memory in both cases is is assumed to be the same, and to operate at the bandwidth of the conventional RISC processor's memory. The y-axis shows the bandwidth available for each byte of available memory (moving from the greatest to the least). Registers are accessed at the clock-speed of the processor (and provide the number of arguments each register file is capable of outputting), and each of the memories is cycled at their minimum access time. The PIM assumes a 2 k-bit register file with 256-bit wide-word access each clock cycle, and a 32 MB memory macro with a 2k-bit random access every 10 ns, typically of memory-in-logic fabrication processes. The superscalar machine has a 4kbit visible register file, is 4-way superscalar, and accesses 4 64-bit registers every clock cycle. It has a 64 KB L1 data cache with 64-byte accesses every 6 clock cycles, and an 8MB L2 data cache with 64-byte accesses every 20 clock cycles. Main memory for both is assumed to provide 2.5 GB/sec of

bandwidth. The cache assumptions are meant to be slightly overly-optimistic for the RISC machine.

The graph shows tremendous potential in the region where the L2 cache in the RISC processor is dominated by the larger, higher bandwidth local PIM memory macro. (A discussion of PIM physical design follows in section 2.2.4.) If this region of high local bandwidth can be exploited, then PIM can potentially provide significant performance improvements over a conventional machine. The traveling thread execution model seeks to do so by treating the local memory region **as memory**, and maximizing the amount of data that can be put there. This differs from the conventional processor that treats all its on-chip memory as a cache (including the software-programmable cache represented by the register file). This local, high-bandwidth region is critical in overcoming the von Neumann bottleneck. Traveling threads attempt a secondary gain in that remote latencies are converted from two-way to one-way transactions.

2.2.4 Microarchitecture Considerations and Memory Layout

As discussed above, the first step in overcoming the von Neumann bottleneck is combining the processor and memory on the same die. Figure 2.4 shows a typical PIM layout. The Memory Macro, as in most memory systems, is laid out in rows and columns. When an address is requested (either for read or write), it is split into two parts: a row address and a column address. The row address is presented to a row decoder, and a signal to read out a particular row is generated on the appropriate *word line* (as indicated in the figure). The entire row (in this case 2K-bits) is presented to the *sense amplifiers* which increase the speed of switching by detecting small changes in the voltage on the *bit lines*. Each column has an associated bit line.

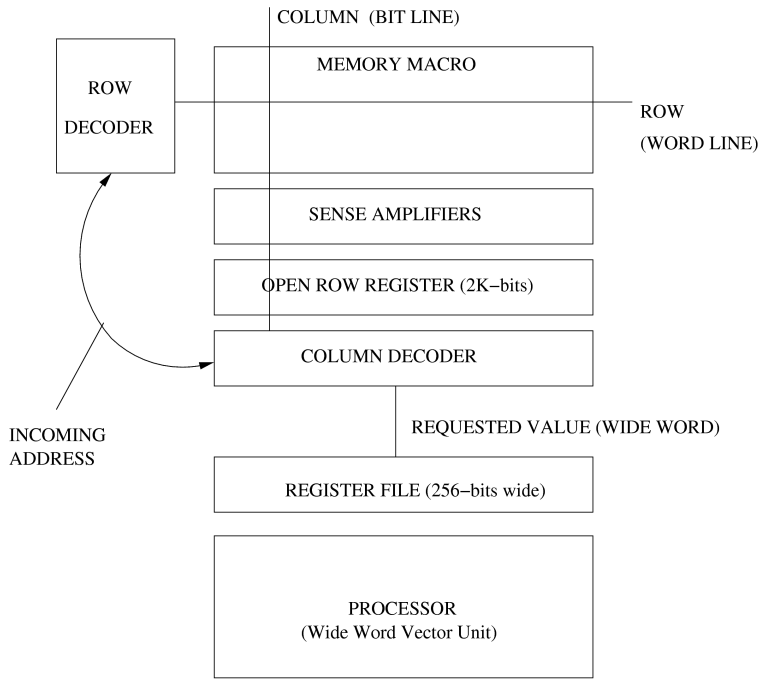


Figure 2.4. Typical PIM Memory Layout

The value from the sense amplifiers is presented to the *open row register*, which contains the *full row* from memory (upwards of 2048 bits). At this point, part of the column portion of the incoming address is used by the *column decoder* to select which wide word (or "page") is read. Since a standard memory part would end here, only a very small portion of the full row can be presented to the memory bus (due to the limitation on the number of pins a chip can have). However, PIMs can take a much larger chunk of the full row (and even the entire full row during some operations). On chip logic, e.g., the processor, is capable of requesting and operating on a *wide word*, typically 128 to 256 bits. All prior PIM microarchitectures have included the ability to exploit these wide-word accesses by treating them as short vectors.

Aside from the fact that a PIM is capable of requesting more data from memory than a normal processor, it is also capable of requesting that data at a significantly

higher rate. By slightly enhancing the column decoder, multiple accesses to the same open row can be served at relatively low cost (since the row can be "latched in" on the first read).

2.2.5 Execution Model Fundamentals: Parcels

PIMs communicate through the use of *parcels*. Parcels are messages possessing intrinsic meaning (e.g., an action to be taken) which are directed at named objects in memory, somewhat similar to active messages (see Section 2.5.3). Rather than merely serving as a repository for data, they carry distinct high-level commands and some of the arguments necessary to fulfill those commands. That command is executed at a processor next to a named memory location, rather than at a named processor. Low level parcels (which may be entirely handled by hardware) may contain simple memory requests such as: "access the value X and return it to node K." Higher level parcels are much more complicated. An example might be "begin execution of procedure Y with the following arguments and return the result to node L." Thus it should be assumed for the rest of this work that a parcel is capable of performing both communication and computation, and it may be generated by the user, run-time system, or hardware by whatever mechanism may be appropriate.

The fact that parcels are directed at named memory objects rather than named processors implies that a structure to determine *where* an object is located is essential to the successful construction of programs using parcels. That is, if any virtualization in the memory exists, the system must be capable of determining where any memory object resides.

Parcels will be used as the basis for constructing the Traveling Thread Execution Model, described in Chapter 4.

2.2.6 Instruction Set Architecture Issues

The machine's Instruction Set Architecture (ISA) is the final step in exposing relevant architectural features to the programmer. The PIM work that this dissertation builds upon has identified several relevant architectural features that connect the technology, the microarchitecture, the system architecture, and the execution model. Section 2.3 discusses the specific PIM systems in more detail, but three general features are extremely important and present in the ISA of each of those systems:

1. **Threads and Explicit Concurrency:** because memory references are high latency activities, the architecture is multithreaded to allow the programmer to identify potentially concurrent operations that the hardware can use to tolerate memory latency. This is reflected in the ISA in the form of thread control instructions (fork, join, etc.), as well as mechanisms that allow for data to be shared between threads that are running concurrently in a producer/consumer ordering.
2. **High Bandwidth Local Accesses:** Because of the microarchitecture of the local memory macros, PIMs have included short-vector operations that can be performed by datapaths that are pitch-matched to the memory. This is reflected in the instruction set as a set of short vector operations capable of operating on different data types pulled from the same 256-bit wide-word (e.g., 32-byte values, 16 16-bit values, 8 32-bit values, etc.) in a SIMD (Single Instruction Multiple Data) or VLIW (Very Long Instruction Word) fashion.
3. **Efficient Remote Communication via Parcels:** Traveling threads build on the parcel model that supports extending remote memory reference semantics to perform more complex operations. In most machines a remote read or write is the only remote operation allowed, but parcels permit remote method invocations, and a number of other actions that include traveling threads. Fundamentally, this is a choice of how to present the underlying communication to the programmer. Chapter 4 discusses the potential instruction set architecture additions necessary to support traveling threads, and Chapter 8 discusses what choices are appropriate in light of the experimentation performed in the rest of this work. However, all prior PIM work has exposed a mechanism to send a parcel and obtain any results that may have been generated by sending the parcel as part of their ISA.

The ISA ties all of the PIM technology, microarchitecture, system architecture, and execution model choices together by presenting to the programmer or compiler a set of fundamental operations that the machine supports. It is the culmination of

the design choices made through experimentation in the rest of the work. Perhaps equally importantly, because it is presented to the programmer and compiler, there is additional difficulty in choosing the semantically correct representation of the operations dictated by the underlying system, and there is difficulty in expressing those operations efficiently (both so that the programmer and compiler will use them and so that the machine can decode and execute them quickly).

2.3 PIM Systems

This section covers several potential PIM system architectures, beginning with two studied at the University of Notre Dame: the Hybrid-Technology Multithreaded (HTMT) project and the Data IntensiVe Architecture Project (DIVA). These projects constitute two diverse system architectures: the former targeted at petaflop scale performance; while the latter was a large workstation accelerating PIM effort, targeted at a teraflop.

2.3.1 HTMT

The HTMT Project [62, 99, 63] was targeted at achieving petaflop-scale performance in the near term by combining numerous advanced technologies. The core processing elements, based on rapid single-flux quantum (RSFQ) devices, operated many orders of magnitude faster than the traditional transistor-based systems constituting the bulk of the memory hierarchy. Consequently, HTMT used the *percolation model* to actively prefetch data required at the RSFQ layer. The primary mechanism used in prefetching consisted of a layer of SRAM and DRAM PIMs actively moving data through the memory hierarchy, performing processing operations on that data, and presenting the RSFQ layer with any computation that requires more heavy-weight processing.

2.3.2 DIVA

DIVA [48, 64] constituted a smaller, more traditional PIM array. A relatively small number of PIMs were used as part of the memory hierarchy of a traditional workstation to provide additional, significant processing support in the memory hierarchy. Fundamentally, this architecture was best suited for scientific applications which exhibit relatively low locality (both spatial and temporal).

2.3.3 BlueGene/C

The original IBM BlueGene initiative (BlueGene/C) [5, 108] proposes using hundreds of thousands of PIMs with extremely simple ISAs to achieve petaflop performance to address the protein folding problem. This system is a true supercomputing use of PIM, but is limited in scope to one specific application (with extensions into a small, similar set of applications). Energy decompositions of the protein folding problem have highly replicable datasets, and the BlueGene architecture is tuned to exploit this fact.

2.3.4 Impulse

The Impulse Memory Controller [30] is a PIM system designed to offload memory management work from the main processor. The controller is capable of performing a diverse set of memory management operations and presenting more "cache friendly" data structures to the processor. For example, it may be used to prefetch sparse vectors, compress them into cache lines, and present them to the processor for some operation.

2.4 Parallel Processor and Memory Architectures

In constructing any parallel system, it is critical to examine the semantics of memory interaction in terms of the memory topology, access methodology, and the

programmatic meaning of memory access.

2.4.1 DSMs and SMPs

The memory systems for parallel machines are often classified as either *Distributed Shared Memory (DSM)* or *Shared Memory Multiprocessors (SMP)*. A DSM, such as the Rice TreadMarks system [10], generally consists of nodes which are tightly coupled to a physical region of memory. The memory on each node is split into *local* and *global* parts, and the shared memory portion consists of the union of the global portion of each node's memory. That is, each node shares the memory it owns with every other node over a network via a software protocol.

Conversely, the nodes of an SMP, such as the Cray MTA [8], or the Sun F15k are tightly coupled to the physical memory in that the memory space is unified by hardware, rather than distributed. That is, there are either separate processing and memory nodes all connected via an interconnection network, or custom memory controllers intervene between the processors and memory locally to provide sharing of data. In either case a load or store instruction executed on any processor can access any location in memory, and the underlying coherency protocol (if any) facilitates the sharing.

PIMs generally fit neither configuration in that while a node consists of both processing and memory elements (as in a DSM), it is conceivable that the processing capability of some nodes may consist of nothing more than servicing requests for memory (as the memory controller would do in an SMP).

2.4.2 Massively Parallel Processors (MPPs)

ASCI Red Storm [28] represents the latest in a long line of highly successful MPPs, including ASCI Red [102]. The system consists of a large number of commodity processors (Opterons in the case of ASCI Red Storm, and Pentiums in the

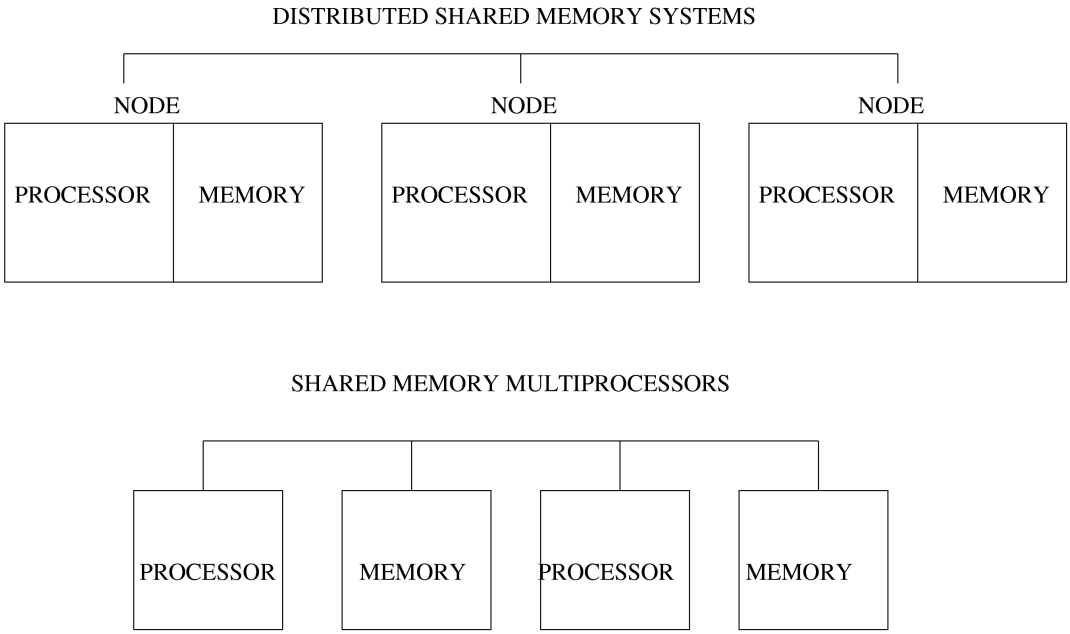


Figure 2.5. DSM and SMP Systems

case of ASCI Red) and memory, connected by a custom Interconnect [7] and running a custom, lightweight Operating System [72, 98, 73]. These systems are not typical "clusters", in that they require extensive custom components and software to scale to tens of thousands of processors, whereas a true commodity cluster is likely limited in scalability to hundreds of processors. They do, however, share the same programming model as clusters, typically MPI[38]. These systems represent the backbone of modern US supercomputing, and typically support an application base devoted to the simulation of physics.

2.4.3 Vector Pipeline Machines

Vector machines, starting with the CDC STAR 100A and the TI ACS, but best typified by the CRAY-1[96], and showing modern incarnations in X1 and Japan's Earth Simulator, provide a latency avoidance mechanism through the use of vectors. The machines have tremendous scatter/gather potential for large, sparse vectors

Figure 2.6. A Typical CC-NUMA implementation

and older version tended to have very good machine balance to combat the von Neumann bottleneck (that is, primary memory was very close to the processor, and programmed via overlays rather than cached). Once the vector operation startup cost had been paid, it is amortized by producing a result from the vector every clock cycle. This poses two potential problems: first, the program must be expressed as vector operations to gain significant speed-up (in fact, scalar operations tend to be much slower than conventional CPUs); and second, because of their high cost, most programmers are not exposed to the programming model of the machine early, unlike an MPP which shares its programming model with inexpensive commodity clusters.

2.4.4 NUMA and CC-NUMA Architectures

Non-Uniform Memory Access (NUMA) machines, as compared to Uniform Memory Access Machines (UMA), share perhaps the most in common with PIM systems, in the sense that local accesses occur quickly while remote accesses may encounter long latencies. In a NUMA architecture supporting a coherency protocol, the programmer has no way of knowing (directly) which memory locations are local or

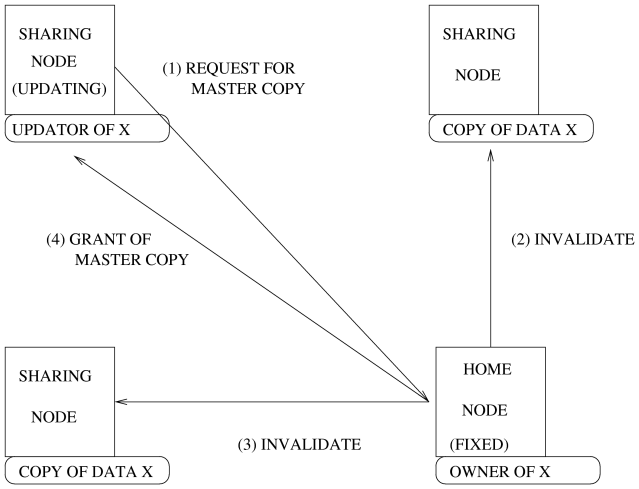
remote.

The memory access on a PIM is non-uniform in the extreme. As previously discussed, local accesses are extremely fast while non-local accesses are likely to be extremely slow. However, this is where the relationship grows increasingly distant. The Cache Coherent NUMA (CC-NUMA) architectures (see Figure 2.6) provide support for coherency amongst nodes in a NUMA system. That is, a *coherency protocol* is defined which ensures that each node contains the semantically correct value of the data which it is examining, which ensures that a consistent computation always takes place.

The Stanford FLASH [67] (which is a descendant of the DASH machine [69]) is a classic example of a CC-NUMA architecture. There is extensive research into both the coherency protocol used and the methods of efficiently storing information as to who is accessing a particular piece of data. The Scalable Coherency Interface (SCI) [57] is an example of a coherency protocol definition (as well as an interconnection network standard). The SCI uses linked lists to track who is accessing a particular cache line. Many other protocols use more expensive sparse matrices. It should be noted, however, that most CC-NUMA organizations do not scale beyond a few thousand nodes because such extensive information must be kept about data which is being shared.

Even though CC-NUMA machines track the sharing of data throughout the system, a given physical address always has a fixed location or *home node*. Although many nodes may be sharing a given address, or even updating that address, when the address is presented to the system it can deterministically be found by referring to the home node.

Figure 2.7 shows the updating of a piece of data under a CC-NUMA scheme. A typical coherency protocol will invalidate shared copies of the data and grant



NOTE: X represents a piece of data which must be updated. The HOME NODE owns X, but is not the node performing the update. The coherency protocol will determine the mechanism for updating data in a semantically correct fashion.

Figure 2.7. A CC-NUMA machine during an update.

master (or update) permission to a single node. The control for this mechanism is accomplished at the home node.

The MIT Alewife [4], the Convex Exemplar and the SGI Origin are other examples of CC-NUMA machines.

2.4.5 COMA Machines

In contrast to CC-NUMA architectures, the Cache Only Memory Architecture (COMA) allows for the actual migration of data throughout the system. That is, a given address does not have a fixed home node, rather the system must determine the home node by looking in a directory. Since the location of data is free to move about the system, addressing simply becomes a name space for data, rather than implying a fixed physical location. When an address is presented to the system, it is located through the use of a *directory*, and it may migrate accordingly. It should be noted that most COMA architectures support Cache Coherent protocols – that

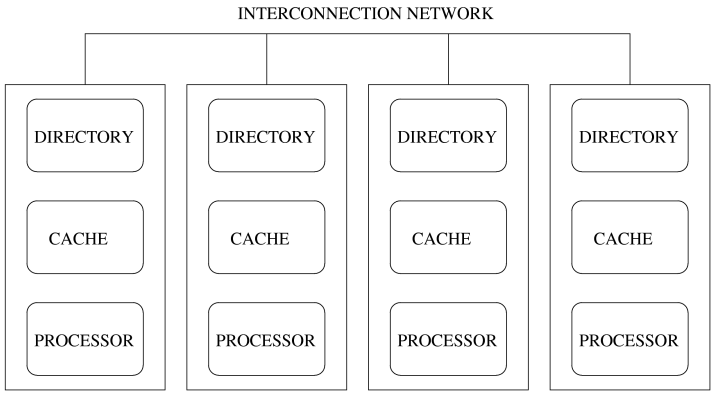
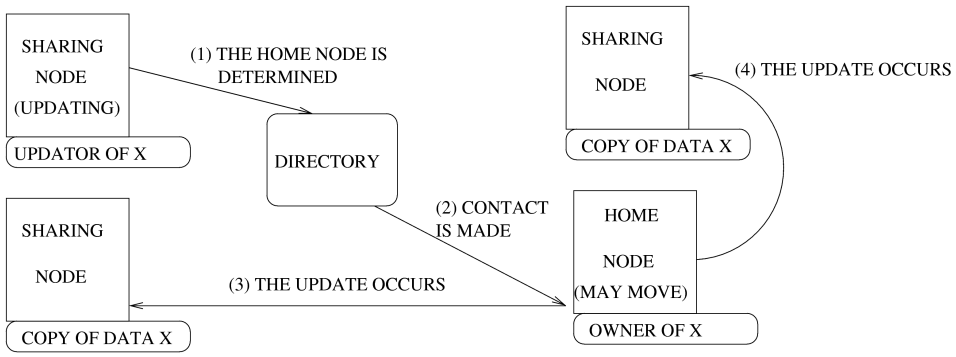


Figure 2.8. A Typical COMA Machine



NOTE: X represents a location/variable that must be updated. The HOME NODE owns X, but is not the node performing the update. The coherency protocol will determine the mechanism for performing the update in a semantically correct fashion. Repeated accesses to X may cause its home node to be moved to the accessing node. Additionally, a directory is required to determine the home node.

Figure 2.9. A COMA machine during an update.

is, there are copies of various pieces of data throughout the system. The primary difference is that the "original" or "master copy" of data may migrate based on demand. See figure 2.8.

Figure 2.9 is an example of a COMA machine performing an update.

There are several examples of COMA architectures constructed in hardware or software including the Data Diffusion Machine (DDM) [106], the Princeton SHRIMP

[36], and Simple COMA [97]. While the construction of the directory to allow the migration of data throughout the system is a complex issue, designers of COMA architectures focus tremendous amounts of time on the cache coherency protocols which allow them to compete with the speed of CC-NUMA machines. Unfortunately, it is this coherency protocol more than the directory structure which limit the scalability of COMA machines.

COMA is essentially traveling threads in reverse, that is the threads reside at fixed locations and the data travels to the appropriate location (without a fixed home node, as in the case of a CC-NUMA machine).

2.4.6 Memory Consistency Models

One of the most critical components that must be addressed here relates to the semantics of memory access. Although the memory sharing mechanisms (as discussed above) are critical, the meaning of one memory access in relation to another requires attention at all levels of the memory hierarchy, including:

- pipeline-to-cache accesses;
- hierarchical cache-to-cache accesses (e.g., L1 to L2);
- peer cache-to-cache or memory-controller to memory-controller access; and
- and, traditional cache-to-memory access.

Memory consistency models [1, 3, 2] specify the programmer's view of shared memory hierarchy. The traditional *sequential consistency* model, in which all memory accesses occur in the order specified by the program, seems the simplest, and more intuitive, however, the hardware and compiler become constrained by the demands of the in-order memory access which enforces uniprocessor semantics on parallel programs. A sequentially consistent memory access model maintains both the program-order of all memory operations issued to a single processor as well as

the corresponding sequential order between operations across all processors. Many commercial processors (the Alpha, UltraSPARC, PowerPC, and MIPS, for example) support *relaxed consistency* models that allows for more out-of-order processing of memory reads and writes. Using a sequential consistency models strongly impacts the coherency protocol required for shared data in two critical ways [41] necessary to support sequential updates:

1. writes must be made visible to all processors; and
2. writes to the same location from the same processor must be *serialized* to maintain consistency.

The first feature critically impacts the scalability of sequentially consistent implementations as it requires either a broadcast during update, or tracking all copies of shared data followed by point-to-point synchronization. As in [1], this work views *cache coherency* protocols as the mechanism by which this serialization is enforced. Typical implementations involve *invalidating* copies of the data once the master copy has been updated. (Updating each copy is also a valid option.) Naturally, this entire process appears atomic to the programmer.

When employing a relaxed consistency model, the simplest set of constraints to relax are those relating to a write followed by a read to an **independent** location in memory. Examples of this optimization include the IBM 370 memory model, *total store ordering*, employed by SPARC V8 processors, and the *processor consistency* model [1]. In these models, reads may be executed out of order with respect to the previous write from the same processor. The three models differ only in the strictness with which the serialization step occurs, with the IBM 370 being the most strict (e.g., requiring write serialization with all other processors before reads may be released), and processor consistency being the least strict (in that any write value may be read before serialization).

Further relaxation of constraints may be achieved by additionally relaxing the constraints between writes. This is known as *partial store ordering* in the SPARC V8 processor[1]. In this mode, a given processor permits writes to differing locations to be overlapped such that they may reach memory out of program order. The programmer may explicitly enforce more strict ordering in this scheme.

Finally, a number of schemes exist for relaxing all program orders, thereby violating sequential consistency. These appear in the DEC Alpha, SPARC V9 (in Relaxed Memory Order), and IBM PowerPC. Weak Ordering provides for explicit programmer synchronization, that is *data* operations are guaranteed no particular ordering unless accompanied by one or more *synchronization* operations. These operations are used to explicitly delineate when reordering would be harmful to program correctness. Similar to weak ordering, the *release consistency* model allows for an additional set of asynchronous semantics, and for explicitly *acquiring* and *releasing* the master copy of a cache line.

Outside of this strictly hardware-oriented framework, other consistency models have been proposed, such as *Location Consistency* [40]. These models tend to require both hardware and compiler, or even programming model changes. It should also be noted that, built upon these hardware memory semantics are a set of *multithreading* memory semantics, including the conventional *mutex* and *semaphore* synchronization mechanisms. The critical point is that thread synchronization typically involves higher level (potentially programmer visible) mechanisms such as conventional locking, and the overall program consistency may be specified explicitly by either the programmer or compiler.

2.5 Threads and Multithreading

The history of multithreading, either in hardware or in software, is long. Multithreading has been used extensively to address the problems of latency and throughput. From the standpoint of software, a *process* is used to encapsulate an address space and a thread of execution. In a system supporting multiple *threads*, the same address space is shared, but more than one instruction stream can be executed in that space (potentially concurrently). In the 1960's, before it was officially identified as multithreading, it was used in the I/O subsystem of many machines (the CDC6600, for example) to manage the latency of accessing storage media. That technique was gradually pulled into the operating system's software for time-sharing systems. Although certainly not the first system to do so, the UNIX time sharing system [94] featured the ability to switch to other processes during long I/O events in the 1970's. The use of multiple processes to tolerate I/O latency is analogous to the use of multiple threads within a single application to tolerate the latency to memory; fundamentally, the difference is in whether or not the two entities (process or thread) share the same address space. In the 1980's, significant work was done to enable applications to perform coarse grained multithreaded operations. Threads scheduled by both the operating system (so called "kernel threads"), and threads scheduled by the application (so called "user threads") were used to tolerate a variety of latencies. Client/server applications exploited the operating system's ability to perform useful work on other processes or kernel threads to mask network and disk latencies, while databases used multiple user threads to increase the throughput of many simultaneous queries. Both showed advantages: while kernel threads required significant time to switch between them, they could be used to tolerate latencies imposed by I/O. User threads, on the other hand, would have to block during long I/O events (since they would have no way to know one occurred),

but could be switched very fast by the user program, and thus helped increase the throughput of computations not encumbered by I/O. Many applications used both types of threads to improve performance.

Supporting threads in processor architectures increased significantly with dataflow machines, such as the Horizon supercomputer[65, 101], the Monsoon[53], and the Cray/Tera MTA[8] (among others). These machines supported switching between a fixed number of threads in hardware. To switch operating system or user threads, the *thread state* (generally consisting of a program counter, stack pointer, and the contents of the processor's register file) would have to be saved and another thread's state loaded. With hardware support for multiple thread contexts, threads could be switched at little or no cost. Consequently, multithreaded machines can issue instructions from different threads to the same pipeline, increasing the overall throughput of threads, and, in some cases, using available threads to tolerate the long latency events of threads that would normally have to wait during memory accesses.

Modern conventional processors have been adopting hardware support for threads to increase aggregate system performance (e.g., throughput) in multiprocess or multithreaded environments. Hyperthreading in Intel processors and multithreaded PowerPC server processors both support multiple threads with the goal of increasing throughput.

The remainder of this section discusses architectures and programming models for multithreaded systems broken down in to broad functional categories: Simultaneous Multithreaded Architectures, or SMTs, which are generally conventional processors that support multiple thread contexts to increase throughput; the Cray/Tera Multithreaded Architecture (MTA), which uses multithreading to tolerate long memory latencies; and Active Messages and The J-Machine, which are

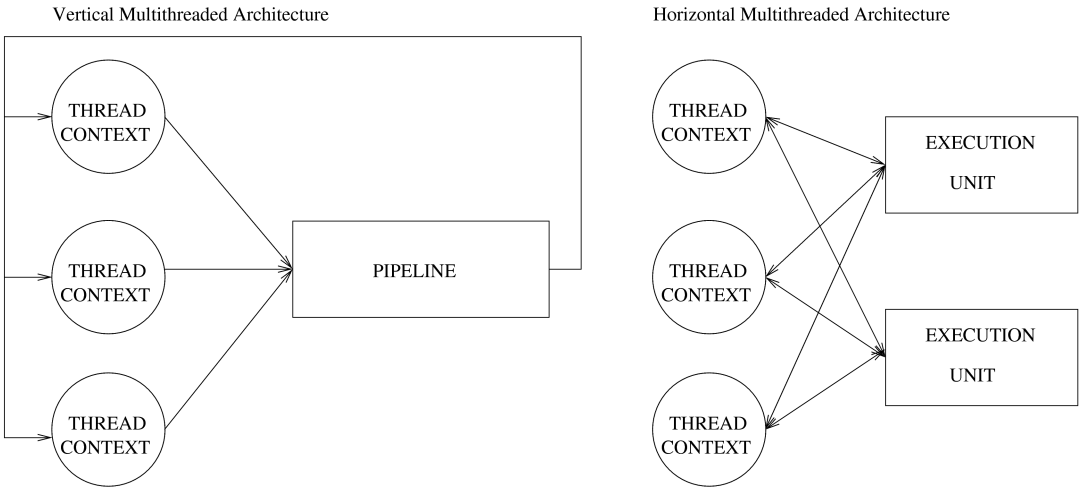


Figure 2.10. Vertical and Horizontal Multithreaded Systems

message passing mechanisms for supporting multithreading, and forms of thread migration.

2.5.1 Simultaneous Multithreaded Architectures (SMTs)

Multithreading extends the traditional process model by allowing more than one execution path in a single address space. In the UNIX model, each process must have at least one thread. Each thread has its own, independent context: a register set and thread-local stack. Communication and synchronization between threads is relatively light weight as all threads in a program share the same address space. Traditionally, threads are scheduled by the user process ("user threads") or the operating system ("kernel threads" or Light Weight Processes).

Multithreaded architectures [75] (MTAs) are typically divided into two categories: *vertical* multithreading, in which the processor can support multiple threads of execution scheduled to a single pipeline; and, *horizontal* multithreading where the processor allows multiple thread contexts to be issued concurrently to its various execution units. The distinction becomes somewhat blurred when one considers

an entire pipeline to be an execution unit. Figure 2.10 summarizes the two architectures. The critical advantage of either methodology is that during high latency operations, the available thread level parallelism can be exploited to keep processing resources busy by quickly switching contexts to any ready-to-run thread when a thread becomes blocked. Context switching may occur as much as every clock cycle to keep the pipeline full. Both categories of MTAs are SMTs in that they allow for multiple threads to be executed by the processor simultaneously.

Some modern architectures billed as multithreaded, however, do not directly exploit thread level parallelism by switching to a new hardware context during cache misses. They purport to achieve high throughput only incidentally. That is, the pipeline does not actively attempt to fill cycles stalled on high latency operations with useful work from another thread. Rather, it simply supports a greater number of thread contexts, and indirectly attempts to lessen the impact of a miss at any given time by scheduling a fixed number of contexts at a time. Specifically, a pipeline consisting of p stages may require $2 \leq n \leq p$ threads to be scheduled. In effect, each thread is given a fixed allocation $\frac{p}{n}$ stages. A thread blocked on a cache miss must consequently inserts pipeline stalls into its $\frac{p}{n}$ pipeline stages for the duration of the miss. Some would argue that this, and other more vertically oriented techniques, are not true multithreaded architectures because they do not directly use *extra* threads for latency toleration, rather their goal is to increase aggregate throughput by executing more than one task at a time. This work will focus on a mix of horizontal and vertical approaches in which many available hardware contexts are exploited in a very fine grain fashion specifically to tolerate both relatively low latency local operations and high latency remote operations.

Multithreading techniques have been exhibited in various forms since the 1960's (even before the term *thread* was coined). The CDC 6600's peripheral processor, for

example, exhibited multithreaded behavior. Modern processors such as server versions of the PowerPC [19] are beginning to support limited multithreading. The Intel Pentium 4's hyper-threading architecture supports a number of hardware threads to mask high latency operations. The Cray (nee Tera) MTA (See section 2.5.2) may be the largest, most relevant system in the context of this work. It should be noted that while all multithreading techniques must support multiple hardware contexts (often in the form of several register files), they are often combined with robust register renaming to take advantage of the instruction level parallelism as well. The interfacing of multithreading to the memory hierarchy can be quite complex. For example, a cache that is shared between two threads from different processors must maintain barriers between those processes. The topics of shared versus separate caches, instruction queuing mechanisms, and multi-core allocation as it relates to thread assignment are all robust research topics. In the context of PIM, however, these design decisions are more strongly coupled to the on-chip memory macro placement, and its relation to the parcel interface.

2.5.2 Cray Multithreaded Architecture

The Cray MTA [8] is the successor of the Horizon [65, 101]. It is a highly parallel machine that seeks to achieve performance by emphasizing throughput and parallelism over speed and complexity.

Each Cray MTA node allows for a zero time cost context switch between threads, which allows for very fine grain thread scheduling. The instruction set is quite simple, and optimized so as to eliminate the need for complex pipelining. Instruction encoding is said to be "horizontal" or moderately Very Long Instruction Word (VLIW) in that each cycle allows for the execution of one memory reference, one ALU operation, and an additional control or ALU operation. There is **no cache**,

TABLE 2.1

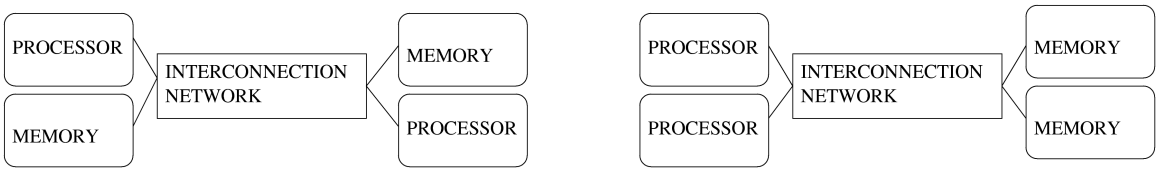
SEMANTICS OF CRAY MTA'S FULL AND EMPTY BITS

VALUE	Meaning on LOAD	Meaning on STORE
0	always read	write then set full
1	reserved	reserved
2	block until full and leave full	wait for full and leave full
3	block until full and set empty	wait for empty and set full

Note: "fullness" indicates data is located in memory and ready to be accessed while "emptiness" indicates that there is no data currently available.

which eliminates the need for a coherency protocol. Because thread contexts can be switched at will, all instructions within a thread are executed in order. Rather than attempting an out of order or speculative execution, the system merely switches focus to another thread whenever there is any event which may cause latency. Most significantly, each node in the system is capable of supporting enough threads so that the latency of a memory accessed may be masked. In fact, the Cray MTA hardware and compiler attempt to trade locality for parallelism whenever possible. All loads are done in the granularity of one 64-bit word, and logically consecutive memory locations are physically distributed throughout the system to eliminate "hot-spots".

A thread context in the MTA consists of one 64-bit stream status word which contains the thread's PC and current machine state, 32 64-bit General Purpose Registers, and 8 64-bit Target Registers used in branching. This means that each thread has 2624 bits of context. There are a maximum of 128 thread contexts available on each node, which means that the register file must hold a total of 41 KB of data. This is similar to the size of an L1 data cache on a typical workstation, but is accessed as a multiported register file, impacting cycle time considerably.



(a) A UNIFORMLY DISTRIBUTED CONFIGURATION

(b) A "DANCE HALL" CONFIGURATION

Figure 2.11. Types of Processor and Memory Distributions

The only synchronization mechanism provided for is through the use of memory. Two "full/empty" bits are provided for each 64-bit word. The semantics of the full/empty bits are described in Table 2.1.

The Cray MTA Interconnection Network is a high speed 3D toroidal mesh, and the machine as a whole is considered an SMP in that there are processing nodes and memory nodes connected to the network. Both types of nodes are intermingled to avoid the typical "dance-hall" configuration of most SMPs where processing and memory elements reside on opposite sides of the network (see Figure 2.11). Again, even in the design of the Interconnection Network, parallelism is emphasized in the extreme.

The success of the Cray MTA is completely dependent upon the ability of the programmer and compiler to provide sufficient parallelism (in the form of enough threads) to keep the system busy at all times. Since locality is of no importance, a large number of threads must be available to run on each node.

In the most recent incarnation of the Cray MTA, the Eldorado, the MTA architecture is combined with the Red Storm interconnection network. This requires merging local memory on each MTA node, making the machine look more like a DSM than an SMP.

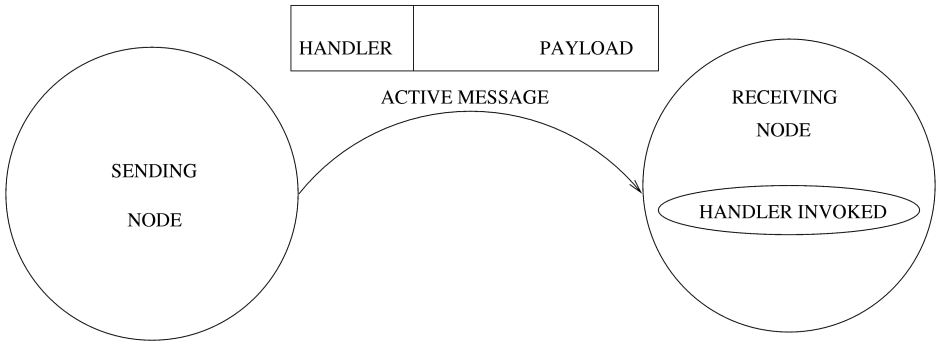


Figure 2.12. Active Messages

2.5.3 Thread Migration: Active Messages and the J-Machine

The topic of thread migration is also a rich research area. A migration occurs any time a thread moves from one node of execution to another. There are numerous mechanisms for doing so, including the ubiquitous remote procedure call (RPC)[17], which was developed as a mechanism to simplify network programming by allowing the application writer to perform remote network transactions by making what appear to be ordinary procedure calls. Effectively, this causes a thread migration from the machine making the the call to the machine performing the call. There is also extremely rich operating systems scheduling work for thread migration in SMP machines that addresses the question of what CPU a blocked thread should be restarted on in terms of "cache affinity", or the likelihood that the CPU's level 2 or 3 cache might contain data the previously blocked thread was processing.

In terms of this work, there are two highly relevant examples of thread migration using message passing: active messages and the J-Machine.

Active Messages [105, 33] seek to maximize the efficiency of parallel machines by minimizing the overhead associated with communication and allowing for the masking of communication latency by overlapping communication and computation.

Rather than a typical message passing system (such as MPI [38]) which provides a mechanism of communication between named processing nodes, Active Messages further allows for the invocation of a message-specified *handler* when a message reaches a given node. The purpose of this handler is two fold: first, it resides in user space, and, given the appropriate system configuration, can spare the application from an expensive call to the Kernel to handle the message; and second, since the handler is invoked upon receipt of the message, systems do not sit idle waiting for communication to occur.

There are several similarities between active messages and parcels. Clearly the notion of invoking some sort of handler (or in parcel parlance, "command") is the same. Parcels, however, are likely to provide increased flexibility in that the invoking of a command does not require that the command be pre-resident on a given node, as is the case with a handler.

This is very similar to work done on the MIT J-Machine [35], which attempted to create an inexpensive massively parallel computer by supporting primitive mechanisms for efficient communication, synchronization and naming of fine grain threads.

2.5.4 Active Pages

Active Pages [84] (developed at UC Davis) attempt to move beyond the von Neumann bottleneck by shifting data intensive computations away from the main processor towards simple PIMs which are constructed from reconfigurable logic. An Active Page is, therefore, the data and its associated functions. These pages are then "processed" by reconfigurable logic (that has precompiled the function). This greatly reduces the traffic between the processor and memory for these applications.

2.6 Programming Models

There are a number of potential programming models to consider deploying on massively parallel PIM arrays. Indeed, this area is a research topic in and of itself. However, when considering PIM deployments, it is critical to be cognizant of the following rational projections:

1. The ratio of processors to memory capacity should be significantly higher on PIM systems. That is, processing resources are readily available.
2. As the size of the array increases (to, potentially, trans-petaflop scale), the increased complexity created by the sheer number of nodes makes shared memory requiring a complex coherency protocol increasingly difficult. This contrasts with more many commercial parallel execution models which assume sharing is feasible and relatively inexpensive (CC-NUMA, COMA, etc.).
3. Program (or thread) code size must be carefully managed to prevent PIM text caching or paging problems. In general, however, code is assumed to be replicable across any node.
4. The PIMs under consideration are pervasively multithreaded to avoid high latency operations at all levels (from the pipeline to inter-node communication). It is necessary that the programming model support the automatic extraction of threads (in addition to those which may be specified by the programmer).

These assumptions often lead to the requirement that the compiler will detect, and adapt to difficult programming constructs, or provide extraordinary support for a particular feature of the architecture. This section will discuss three potential programming models: the imperative model, which dominates modern computer languages, and must be supported; the functional model, which may provide additional levels of concurrency; and, finally, actor-based (imperative or functional) models which map well onto the basic traveling thread and communication mechanisms provided by the PIM.

2.6.1 Imperative Models

Imperative programming may be classified as the most familiar model for computer scientists. The most fundamental aspect of the model, as compared to the

Functional Model described in Section 2.6.2, is that the imperative model allows for location reuse (or multiple assignment). This mechanism allows for *side effects*, which cause increased complexity when performing program dependency analysis, while at the same time providing a useful mechanism for critical systems functions (I/O and temporal sequencing being the most obvious examples). As an example, consider the following C code:

```
int x=0;                /* a global variable */

int f(int y)
{
    x = x + y;
}
```

Although the example is trivial, it demonstrates the side effects inherent in imperative languages. Each time `f` is called, the value of x is updated, and may be used by another function elsewhere in the program. The sequencing of the updating of `x` (through the call to `f`) cannot be automatically parallelized by the compiler without a global understanding of how the updates can occur. That is, it must occur in the order specified by the programmer to ensure "correctness".

2.6.2 Functional Models

By contrast, the single-assignment oriented functional model may provide additional opportunities to exploit concurrency, as suggested in Backus' Turing Award Lecture[13]. These models are primarily based upon the *lambda calculus* [60]. In functional languages directly based upon lambda calculus, such as Lisp or Scheme, the language itself is syntactically simply an extended version of lambda calculus. Consider the Scheme definition of a function, f, which takes two arguments x and y, and implements the equation $x + 2xy$:

```
(define f ((lambda x y) (+ (x (* 2 (* x y))))))
```

Once that function is given arguments, it can be legally evaluated in any order. That is, if `x=37` and `y=42`, then the function can be legally written as:

```
(+ (37 (* 2 (*37 42))))
```

That expression may then be evaluated in any order because it is side effect free. This simple fact means that all parts of this expression (and every other expression in the program) can be evaluated in parallel. Although the syntax of other functional languages, such as ML, may not be based upon lambda calculus, the core program execution model is. Thus, the ML version of the function given above, `fun f(x,y) = x + y;` will be reduced to the same internal representation as the Scheme function. Specifically, all lambda calculus functions can be reduced into *combinator graphs* [60] through the process of *bracket abstraction*. This graph can consist of as few as two combinators (`S` and `K`) and can express any computation that can be executed on a Turing machine. As with the expression substitution discussed above, combinator graphs can be reduced in any order, providing a tremendous amount of concurrency. The application of combinator graphs in parallel architectures is discussed in [81].

Because of the single assignment inherent in functional languages, a combinator graph can be evaluated in any order, and at any level of granularity (e.g., it can be divided into arbitrarily small size, down to single-combinator evaluation, and the pieces can be executed in parallel). Consequently, in a parallel machine synchronization and sharing pressure is significantly reduced. This would make a highly concurrent model for traveling threads.

2.6.3 Actor Based Languages

Actor Based Languages, such as Smalltalk [42] and Objective C [82], provide a unique match to the parcel interface discussed in Section 2.2.5. These object

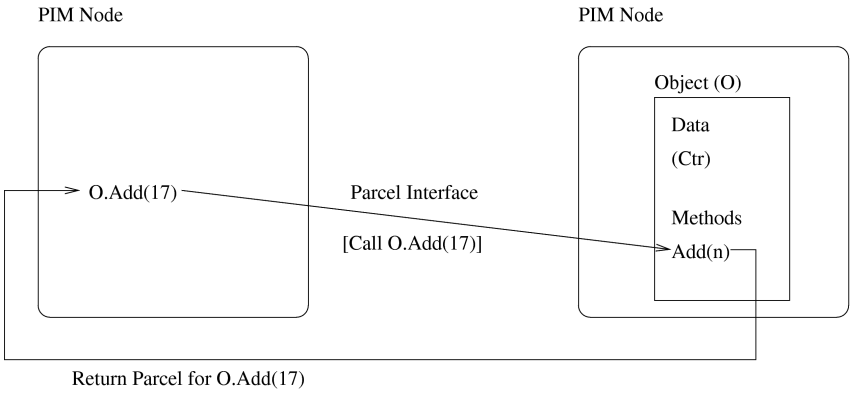


Figure 2.13. Actor Based Language Message Passing

oriented languages provide a "natural" clustering mechanism for data and code in that when a particular object is instantiated, its data and methods can be co-located. Critically, unlike C++, these languages are pure object-oriented languages in that the mechanism for calling a method associated with a particular instantiation of an object is a message sent to that object. This mechanism maps well onto large PIM arrays for two reasons:

1. method calling between two objects located on different nodes can be accomplished by sending a parcel from one object to another with the appropriate argument;
2. the objects can be automatically distributed among different PIMs

Figure 2.13 depicts an example communication to object `O`, requesting that it execute its method `Add` with the argument `n=17`. The communication occurs as the result of the program's structure, rather than being specified by the programmer, generated by the runtime system, or extracted by the compiler via complex dependency analysis.

2.7 Conclusions and Architectural Relevance to PIM

Processing-in-memory provides a unique set of advantages for overcoming the von Neumann bottleneck, in that there is a large amount of on-chip memory available to contain the working set of a particular set of threads. Combining a large number of PIMs into a trans-petaflop scale computer, however, is very difficult. It is well known that coherency protocols are expensive. Both CC-COMA and CC-NUMA architectures are not considered scalable beyond a few thousand nodes. The multithreaded advantages upon which the Cray MTA relies to achieve performance is indeed provocative since a simple synchronization mechanism can be substituted for a full coherency protocol. Providing sufficient threads on a system of 1,000,000 or more nodes, however, may prove impossible. Multithreading can still be used to mask tremendous amounts of latency and provide high node utilization, however it is critical that the the large local memory space of a PIM be utilized effectively.

In terms of communication, parcels allow for many of the features of Active Messages. Rather than being constructed on top of commodity workstations, however, Parcels are integrated with the architecture and multithreading system. Given this tight integration of processing resources, memory, and communication, it is critical to understand the impact of multiple programming models on the architecture.

CHAPTER 3

BENCHMARKS

Throughout this work two sets of benchmarks are examined: first, a set of typical scientific applications from Sandia National Labs, which represent a real class of Floating Point Benchmarks; and second, a set of Integer Benchmarks, primarily from Sandia with some additional public codes included, largely from graph theory, which represent an emerging class of very sparse, memory intensive, discrete math applications that are increasingly interesting in the supercomputing domain. The two suites are different in character: the Sandia floating point suite represents long established large-scale scientific codes, while the integer suite, with a few exceptions, tends to be primarily single algorithm codes, typical of true "benchmarks". Of particular importance for this work are the memory characteristics and the potential concurrency available in the application suite. In order to quantify the difference between typical computer architecture benchmarks and the benchmarks under study, these benchmarks are compared to the SPEC CPU 2000 suite. This chapter demonstrates that both new sets of benchmarks are significantly more memory intensive because of the nature of the applications, rather than their performance on any particular architecture.

Section 3.1 summarizes how the benchmarks were traced and analyzed. Sections 3.2 and 3.3 describe the floating point and integer benchmarks under study in this dissertation. Section 3.4 covers the SPEC CPU 2000 suite that is used in

this chapter to establish the critical differences in performance between the chosen benchmarks and those typically studied by computer architects. Section 3.5 explains how aggregate performance numbers for each suite are computed. Section 3.6 provides an analysis of the *temporal working set*, and Section 3.7 examines the *spatial locality* characteristics of the applications. The *temporal working set* is defined as the number of words actively being used by the application, and the *spatial locality* represents the number of unique words touched over a given interval of instructions, and is clustered into various block sizes. This allows for the analysis of the memory reference properties of the application, as opposed to a typical "cache study" which provides the trade-offs for a set of specific cache implementations. Finally, Section 3.8 enumerates the conclusions that can be drawn from these comparisons.

3.1 Methodology

Each benchmark was traced using the Amber instruction trace generator described in Section 2.1.1. Trace files containing 4 billion sequential instructions sampled from critical sections of the program were stored for later analysis, which provides the same inputs to every analysis tool. This yields a slightly more exacting level of comparison as opposed to execution based simulation, such as SimpleScalar[23], which have minor variations between runs of the program (e.g., due to calls to random number generators, slightly different sampling, or minor differences in system call execution).

The instruction interval starting point was chosen using a combination of performance register profiling of the memory system, code reading, and, in the case of SPEC accumulated knowledge of good sampling points. The 4 billion instruction sequences typically represent multiple executions of the main loop (multiple time steps for the Sandia floating point benchmarks).

3.2 Floating Point Benchmarks

Real scientific applications tend to be significantly different from common processor benchmarks, such as the SPEC suite. Their datasets are significantly larger, the applications themselves are much more complex, and they are designed to run on large-scale machines. The following benchmarks were selected to represent critical problems in supercomputing seen by the largest scale deployments in the United States. The input sets were all chosen to be representative of real problems, or, when they are benchmark problems, are the typical performance evaluation benchmarks used during new system deployment. Two of the codes are benchmarks, sPPM (see Section 3.2.4), which is part of the ASCI 7x benchmark suite, and Cube3 which is used as a simple driver for the Trilinos linear algebra package. The sPPM code is a slightly simplified version of a real-world problem, and, in the case of Trilinos, linear algebra is so fundamental to many areas of scientific computing that studying core kernels is significantly important.

All of the codes are written for MPPs using the MPI programming model (see Section 2.4.2), but, for the purposes of this study, were traced as a single node run of the application. Even without the use of MPI the codes are structured to be MPI-scalable. Other benchmarking (both performance register and trace-based) has shown that the local memory access patterns for a single node of the application and serial runs are substantially the same.

In the case of benchmark problems, such as the CTH 2-Gas problem (see Section 3.2.2, the problems were scaled to single-node size (e.g., with a memory footprint of approximately 2 GB on the node under study), with the help of experts in the particular application.

3.2.1 LAMMPS

LAMMPS represents a classic molecular dynamics simulation designed to represent systems at the atomic or molecular level[89, 90]. The program is used to simulate proteins in solution, liquid crystals, polymers, zeolites, and simple Lenard-Jones systems. The version under study is written in C++, and 2 significant inputs were chosen for analysis:

1. *Lenard Jones Mixture*: This input simulated a 2048 atom system consisting of three different types;
2. *Chain*: simulates 32000 atoms and 31680 bonds.

LAMMPS consists of approximately 30,000 lines of code.

3.2.2 CTH

CTH is a multi-material, large deformation, strong shock wave, solid mechanics code developed over the last 3 decades at Sandia National Laboratories. CTH has models for multi-phase, elastic viscoplastic, porous and explosive materials. CTH supports multiple types of meshes:

- Three-dimensional rectangular meshes;
- two-dimensional rectangular and cylindrical meshes; and
- one-dimensional rectilinear, cylindrical, and spherical meshes.

It uses second-order accurate numerical methods to reduce dispersion and dissipation and produce accurate, efficient results. CTH is used extensively within the Department of Energy laboratory complexes for studying armor/anti-armor interactions, warhead design, high explosive initiation physics and weapons safety issues. It consists of approximately 500,000 lines of Fortran and C.

CTH has two modes of operation: with or without adaptive mesh refinement (AMR)[1]. Adaptive mesh refinement changes the application properties significantly and is useful for only certain types of input problems. One AMR problem and two non-AMR problems were chosen for analysis.

Three input sets were examined:

1. **2-Gas:** The input set uses an $80 \times 80 \times 80$ mesh to simulate two gases intersecting on a 45 degree plane. This is the most "benchmark-like" (e.g., simplified) input set, and is included to better understand how representative it is of real problems.
2. **Explosively Formed Projectile (EFP):** The simulation represents a simple Explosively Formed Projectile (EFP) that was designed by Sandia National Laboratories staff. The original design was a combined experimental and modeling activity where design changes were evaluated computationally before hardware was fabricated for testing. The design features a concave copper liner that is formed into an effective fragment by the focusing of shock waves from the detonation of the high explosive. The measured fragment size, shape, and velocity is accurately (within 5%) modeled by CTH.
3. **CuSt AMR:** This input problem simulates a 4.52 km/s impact of a 4 mm copper ball on a steel plate at a 90 degree angle. Adaptive mesh refinement is used in this problem.

3.2.3 Cube3

The Cube3 benchmark is meant to be a generic linear solver and drives the Trilinos[52] frameworks for parallel linear and eigensolvers. Cube3 mimics a finite element analysis problem by creating a beam of hexagonal elements, then assembling and solving a linear system. The problem can be varied by width, depth, and degrees of freedom (e.g., temperature, pressure, velocity, or whatever physical modeling the problem is meant to represent). The physical problem is 3 dimensional (width $\times$ width $\times$ depth). The number of equations in the linear system is equal to the number of nodes in the mesh multiplied by the degrees of freedom at each node. There are two variants based on how the sparse matrices are stored:

[1] AMR typically uses graph partitioning as part of the refinement, two algorithms for which are part of the integer benchmarks under study. One of the most interesting results of including a code like CTH in a "benchmark suite" is its complexity

1. **CRS:** a 55x55 sparse compressed row system; and
2. **VBR:** a 32x16 variable block row system.

These systems were chosen to represent a large system of equations.

3.2.4 sPPM

The sPPM benchmark is part of the ASCI Purple benchmark suite as well as the 7× application list for ASCI Red Storm. It solves a 3D gas dynamics problem on a uniform Cartesian mesh using a simplified version of the PPM (Piecewise Parabolic Method) code. The hydrodynamics algorithm requires three separate sweeps through the mesh per time step. Each sweep requires approximately 680 flops to update the state variables for each cell. The sPPM code contains over 4000 lines of mixed Fortran 77 and C routines. The problem solved by the sPPM involves a simple, but strong (about Mach 5) shock propagating through a gas with a density discontinuity.

3.3 Integer Benchmarks

While floating point applications represent the classic supercomputing workload, problems in discrete mathematics, particularly graph theory, are becoming increasingly prevalent. Perhaps most significant of these are fundamental graph theory algorithms. These routines are of fundamental importance in the fields of proteomics, genomics, data mining, pattern matching and computational geometry (particularly as applied to medicine) and have direct national security implications. Furthermore, their performance emphasizes the critical need to address the von Neumann bottleneck in a novel way. The data structures in question are very large, sparse, and referenced *indirectly* (e.g., through pointers) rather than as regular arrays. Despite their vital importance, these applications are significantly underrepresented

in computer architecture research, and there is currently little joint work between architects and graph algorithms developers.

In general, the integer codes are more "benchmark" problems (in the sense that they use non-production input sets), heavily weighted towards graph theory codes, than are the floating point benchmarks.

3.3.1 Graph Partitioning

There are two large-scale graph partitioning heuristics included here: Chaco[50] and Metis[59]. Graph partitioning is used extensively in automation for VLSI circuit design, static and dynamic load balancing on parallel machines, and numerous other applications. The input set in this work consists of a 143,437 vertex and 409,593 edge graph was partitioned into 1,024 balanced parts (with minimum edge cut between partitions).

3.3.2 Depth First Search (DFS)

DFS implements a Depth First Search on a graph with 2,097,152 vertices and 25,690,112 edges. DFS is used extensively in higher-level algorithms, including identifying connected components, tree and cycle detection, solving the two-coloring problem, finding Articulation Vertices (e.g., the vertex in a connected graph that, when deleted, will cause the graph to become a disconnected graph), and topological sorting.

3.3.3 Shortest Path

Shortest Path computes the shortest path on a graph of 1,048,576 vertices and 7,864,320 edges, and incorporates a breadth first search. Extensive applications exist in real world path planning and networking and communications.

3.3.4 Isomorphism

The graph isomorphism problem determines whether or not two graphs have the same shape or structure. Two graphs are isomorphic if there exists a one-to-one mapping between vertices and edges in the graph (independent of how those vertices and edges are labeled). The problem under study confirms that two graphs of 250,000 vertices and 10 million edges are isomorphic. There are numerous applications in finding similarity (particularly, subgraph isomorphism) and relationships between two differently labeled graphs.

3.3.5 BLAST

The Basic Local Alignment Search Tool (BLAST)[6] is the most heavily used method for quickly searching nucleotide and protein databases in biology. The algorithm attempts to find both local and global alignment of DNA nucleotides, as well identifying regions of similarity embedded in two proteins. BLAST is implemented as a dynamic programming algorithm.

The input sequence chosen was obtained by training a hidden Markov model on approximately 15 examples of piggyBac transposons from various organisms. This model was used to search the newly assembled aedes aegypti genome (a mosquito). The best result from this search was the sequence used to search in the blast search. The target sequence obtained was blasted against the entire aedes aegypti sequence to identify other genes that could be piggyBac transposons, and to double check that the subsequence is actually a transposon.

3.3.6 zChaff

The zChaff program implements the Chaff heuristic[76] for finding solutions to the Boolean satisfiability problem. A formula in propositional logic is *satisfiable* if there exists an assignment of truth values to each of its variables that will make

TABLE 3.1

SPEC CPU2000 INTEGER SUITE

Benchmark	**Programming Language**	**Description**
164.gzip	C	Data Compression
175.vpr	C	Placement and Routing for FPGAs
176.gcc	C	GNU C Compiler
181.mcf	C	Combinatorial Optimization
186.crafty	C	Chess
197.parser	C	Word Processing
252.eon	C++	Visualization
253.perlbmk	C	PERL
254.gap	C	Group Theory
255.vortex	C	Object Oriented Database
256.bzip2	C	Data compression
300.twolf	C	VLSI Placement and Routing

the formula true. Satisfiability is critical in circuit validation, software validation, theorem proving, model analysis and verification, and path planning. The zChaff input comes from circuit verification and consists of 1,534 Boolean variables, 132,295 clauses with 5 instances, that are all satisfiable.

3.4 SPEC

The SPEC CPU2000 suite is by far the most (currently) studied benchmark suite for processor performance[32]. This dissertation uses both the SPEC-Integer and SPEC-FP components of the suite, as summarized in Tables 3.1 and 3.2 respectively, as its baseline comparison for benchmark evaluation.

3.4.1 SPEC Integer Benchmarks

The SPEC Integer Suite, summarized in Table 3.1, is by far the most studied half of the SPEC suite. It is meant to generally represent workstation class problems. Compiling (176.gcc), compression (164.gzip and 256.bzip2), and systems administration tasks (253.perlbmk) have many input sets in the suite. These tasks tend to be somewhat streaming on average (the perl benchmarks, in particular, perform a lot of line-by-line processing of data files). The more scientific and engineering oriented benchmarks (175.vpr, 181.mcf, 252.eon, 254.gap, 255.vortex, and 300.twolf) are somewhat more comparable to the Sandia integer benchmark suite studied throughout the rest of this work. However, selectively choosing benchmarks from SPEC produces generally less accurate comparisons than using the entire suite (although it would lessen the computational requirements for analysis significantly).

It should be noted that the SPEC suite is specifically designed to emphasize computational rather then memory performance. Indeed, other benchmark suites, such as the Streams [74] suite or Giga-Updates Per Second (GUPS) focus much more extensively on memory performance. However, given the nature of the memory wall, what is important is a mix of the two. SPEC, in this work, represents the baseline only because it is, architecturally, the most studied benchmark suite. Indeed, a benchmark such as GUPS would undoubtedly overemphasize the memory performance at the expense of computation, as compared to the real-world codes in the Sandia suite.

3.4.2 SPEC Floating Point Benchmarks

The SPEC Floating Point suite is summarized in Table 3.2, and primarily represents scientific applications. At first glance, these applications would appear very similar to the Sandia Floating Point suite, however the scale of the applications (in

TABLE 3.2

SPEC FLOATING POINT SUITE

Benchmark	**Programming Language**	**Description**
168.wupwise	Fortran 77	Physics - Quantum Chromodynamics
171.swim	Fortran 77	Shallow Water modeling
172.mgrid	Fortran 77	Multi-grid Solver
173.applu	Fortran 77	Parabolic PDEs
177.mesa	C	3d Graphics
178.galgel	Fortran 90	Computational Fluid Dynamics
179.art	C	Adaptive Resonance Theory Neural Net
183.equake	C	Seismic Wave Propagation
187.facerec	Fortran 90	Face Recognition
188.ammp	C	Computational Chemistry
189.lucas	Fortran 90	Primary Number Testing
191.fma3d	Fortran 90	Finite Element Crash Simulation
200.sixtrack	Fortran 77	High Energy Physics Accelerator Design
301.apsi	Fortran 77	Meteorology: Pollutant Distribution

terms of execution time, code complexity, and input set size) differs significantly.

3.5 Mean Performance Computation

To best summarize the benchmark suites under study, each of the measures gathered experimentally include a "mean result" computed for a given benchmark suite over the input sets. In each case, the mean is computed by the same method. Since some programs in each suite contain multiple input sets, computing a uniformly weighted mean for each run tends to overemphasize those benchmarks. Consequently, each mean is computed as the weighted sum *for each program* in the suite (rather than each run). For example, in the case of SPEC-FP, there are 14 programs

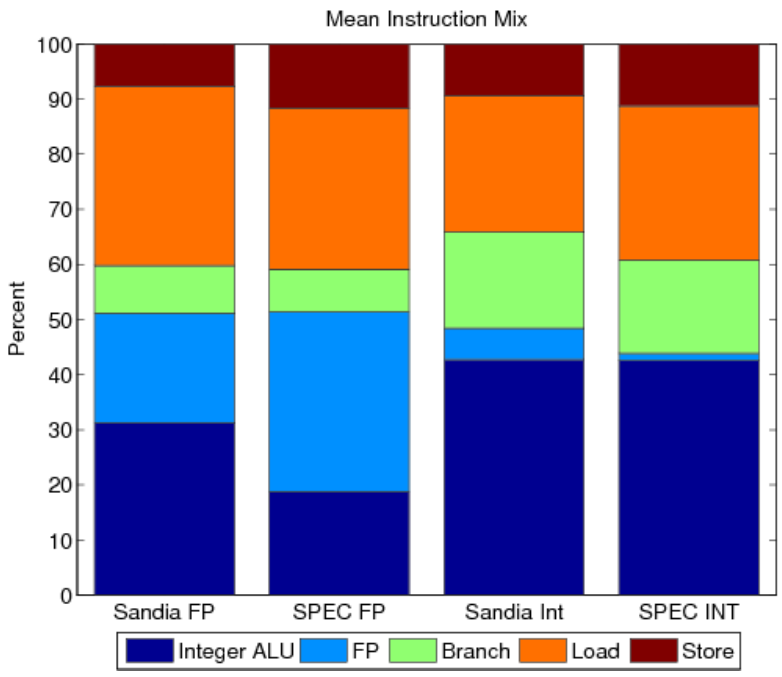


Figure 3.1. Benchmark Suite Mean Instruction Mix

and 15 runs (179.art has two input sets), consequently each 179.art input set only contributes $\frac{1}{28}$ ($\frac{1}{2}$ of $\frac{1}{14}$) to the mean.

3.5.1 Initial Observations of Program Characteristics

Figure 3.1 shows the instruction mix breakdown for the benchmark suites. A detailed mix for each benchmark can be found in Appendix A, Figure A.1. Of particular importance is that the Sandia Floating Point applications perform significantly more **integer** operations than their SPEC Floating Point counterparts, in excess of 1.66 times the number of integer operations, in fact.

Figure 3.2 shows the ratio of integer instructions to floating point instructions. There is an obvious discrepancy in integer operations performed by the Sandia floating point suite as opposed to SPEC FP. In fact, the Sandia suite has a median integer to floating point ratio of 3.95 as compared to SPEC FP's 0.7075. This is largely due

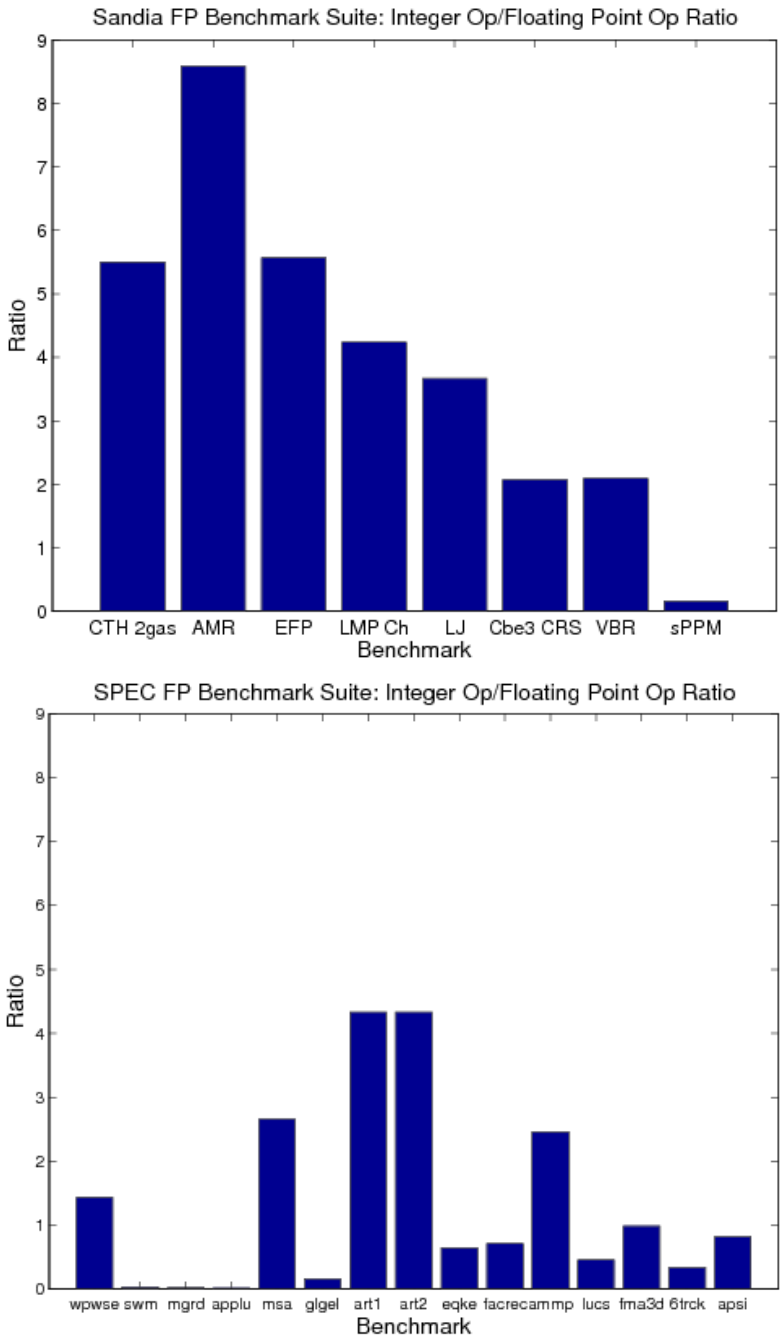


Figure 3.2. Integer Instruction to Floating Point Instruction Ratios for the Sandia and SPEC FP Suites

TABLE 3.3

SANDIA INTEGER APPLICATIONS WITH SIGNIFICANT FLOATING POINT COMPUTATION

Application	**Percent Floating Point Instructions**
Chaco	15.84%
DFS	14.74%
Isomorphism	13.41%

to the complexity of the Sandia applications (with many configuration operations requiring integer tests, table look ups requiring integer index calculations, etc.) as well as their typically more complicated memory addressing patterns. Additionally, in the case of the floating point applications, although the Sandia applications perform only about 1.5% more total memory references than their SPEC-FP counterparts, the Sandia codes perform 11% more loads, and only about $\frac{2}{3}$ the number of stores, indicating that the results produced require more memory inputs to produce fewer memory outputs. The configuration complexity can also be seen in that the Sandia codes perform about 11% more branches than their SPEC counterparts.

In terms of the integer applications, the Sandia codes perform about 12.8% fewer memory references over the same number of instructions, however Sections 3.6 and 3.7 will demonstrate that those references are significantly harder to capture in a cache. The biggest difference is that the Sandia Integer codes perform 4.23 times the number of floating point operations as their SPEC Integer counterparts. This is explained by the fact that three of the Sandia Integer benchmarks perform somewhat significant floating point computations.

Table 3.3 summarizes the three Sandia Integer Suite applications with significant floating point work: Chaco, DFS, and Isomorphism. Their floating point ratios are quite below the median for SPEC FP (28.69%), but above the Sandia Floating Point median (10.67%). They are in the integer category because their primary computation is an integer graph manipulation, whereas CTH is in the floating point category even though runs have a lower floating point percentage (a mean over its 3 input runs of 6.83%), but the floating point work is the primary computation. For example, Chaco is a multilevel partitioner and uses spectral partitioning in its base case, which requires the computation of an eigenvector. In either case, the SPEC FP suite is significantly more floating point intensive.

3.6 Temporal Working Set Characteristics

Rather than presenting the results from yet another cache study, this experiment attempts to quantify the memory system characteristics of the applications under study by measuring the *temporal working set*, which is defined as the amount of data (or memory) actively being used by the program during a particular phase of computation. This measure is orthogonal to the *spatial locality* measure presented in Section 3.7. Conceptually, the temporal working set miss rate data is generated by constructing a 128 MB, fully associative, true LRU cache with a block size of 4 bytes (the native word size on the 32-bit PowerPC traces being analyzed). The working set is modeled conceptually as a 32M entry doubly linked list (though the data structure used in this experiment is optimized to improve performance, it retains the same ordering). During each load or store operation, the list is searched for the requested word address. If a hit is found, the position in the list is used as the block number, and a hit counter for that block is incremented. (The head of the list is position 1, and the tail is position 32M.) That entry in the list is then promoted to

position one in the list, as it was the most recently used item. The hit counters for each word in the cache are used to provide miss rate information for working sets varying in size from 1 4-byte block to 32M 4-byte blocks.

Each block represents a contiguous piece of memory, and is determined exactly as it would be by a caching memory system. Specifically, the first block begins at memory location zero, and subsequent blocks are numbered from the first. For example, a 4-byte block size would have the first block representing memory bytes 0-3, the second bytes 4-7, the third 8-11, and so on. For a block size of 2^k, the block number can be determined by making the k lower-order bits of any memory address 0.

For performance purposes, the list is divided into multiple sections, and a balanced red-black tree is used to locate which section a given address is in quickly. That section is then searched linearly for the address so that the position within the list can be found. The list in this experiment consisted of 16K sections, each of 2K entries. Promotion to the head of the list can be accomplished in constant time, and searching in log time for the section number, and linear time within the section. Given the size of the list, this performance enhancement is required to allow for a reasonable run time.

3.6.1 Miss Rate Interpretation

Because the temporal working set's miss rate does not directly translate into the traditional interpretation of a cache miss rate, it is critical to understand the meaning of various points on the temporal working set miss rate curve. These points are analogous to the same points on a standard cache miss rate curve (showing miss rate vs. cache size), except that the working set curve strictly represents temporal locality. It is equivalent to a cache miss rate vs. cache size curve for a true LRU

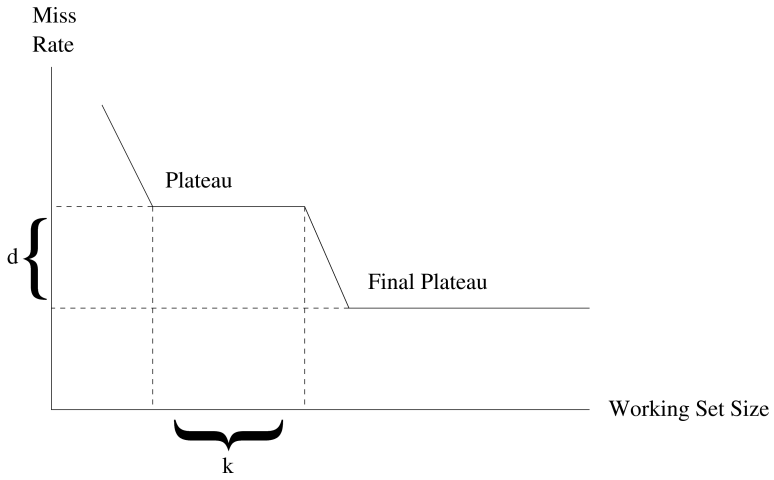


Figure 3.3. Working Set Miss Rate Interpretation

cache with 4-byte blocks at the given cache size. The working set miss rate curve is best thought of as the minimum bandwidth required by the temporal working set of a program given a cache size (because each block in the modeled cache is a single word, there all of the data read is consumed). The necessary bandwidth will increase as the block size increases if all the words in the block are unused. Section 3.7 discusses the overhead of increased cache block size in detail.

Figure 3.3 represents an annotated working set miss rate curve. The miss rate, as expected, is simply the percent of accesses that are not held in a working set of a given size. When the miss rate plateaus, additional growth in the size of the working set has no effect on the **number** of useful items captured. The final plateau represents the *compulsory miss rate*, or the probability that a given memory access over the interval will read truly new data.

The difference (d in Figure 3.3) between the miss rate at the final plateau and the miss rate at a given point represents the probability that any given access is examining "old" data not contained in the working set (e.g., that a smaller working set will have evicted due to size constraints). Similarly, k on the figure represents

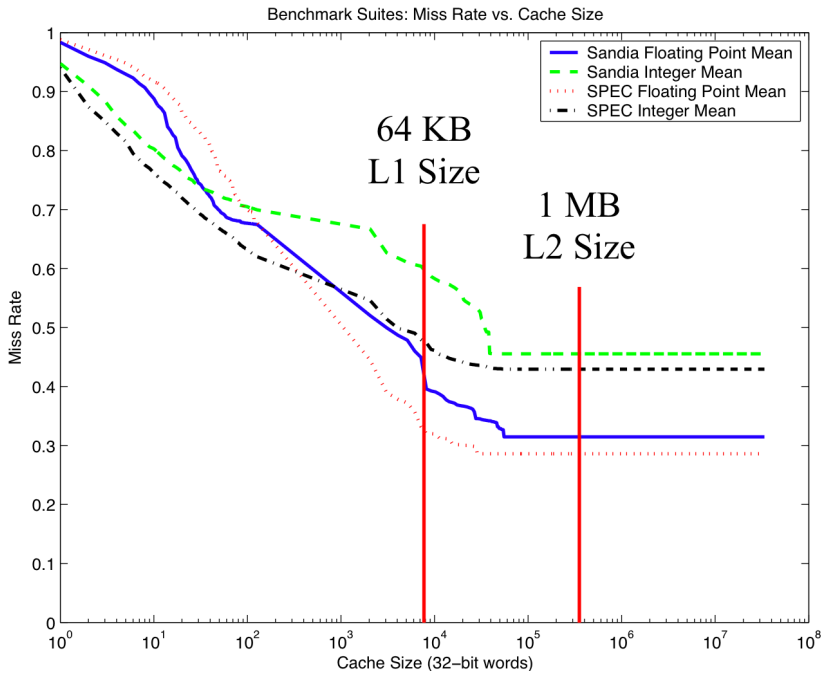


Figure 3.4. Mean Working Set Miss Rates

the increase in working set size required to overcome a plateau (that is essentially wasted until greater than k blocks are added).

3.6.2 Results

Figure 3.4 shows the temporal working set results averaged for each benchmark suite (the full results can be found in Appendix A, Figure A.2). Although the working set miss rates look close, some of the largest gaps occur in the Level 1 cache size range (32-64KB). Both the Sandia integer and floating point benchmarks have between a 20% and 28% increase in their miss rates from their SPEC counterparts. In Level 2 cache sizes (1-8MB), the Sandia integer benchmark suite averages 10% worse, and the floating point benchmarks average 6% worse.

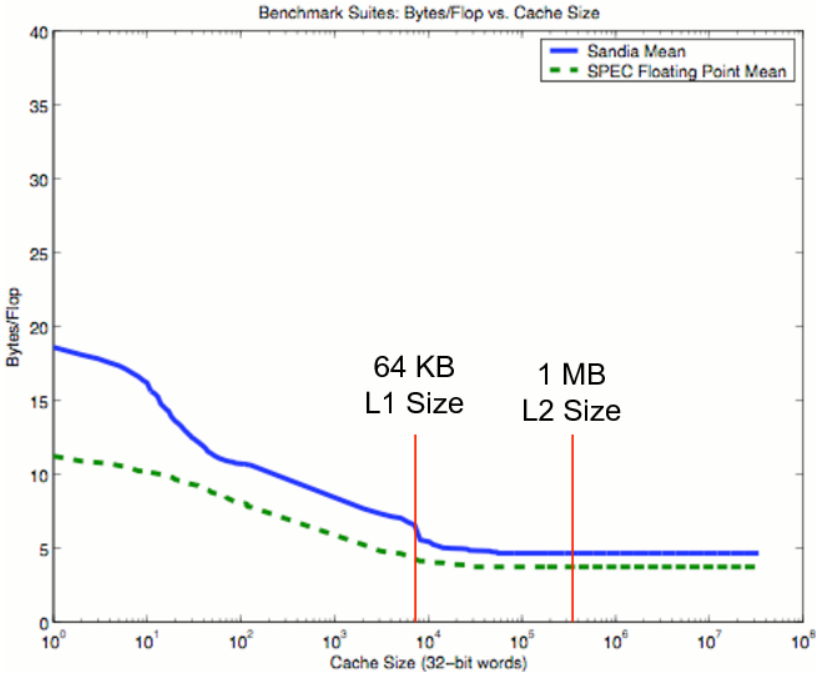


Figure 3.5. Mean Working Set Memory Bytes Consumed per Flop Executed

3.6.3 Bandwidth per Flop

Because the floating point suite was chosen on the basis of floating point operations being the primary work in the application, measuring bandwidth required of the memory system in terms Bytes per floating point operation (flop) fundamentally demonstrates the amount of memory work required per key unit of computation. Specifically, for each unit of work (a flop), this experiment shows the demand placed on the memory system system (as measured by bandwidth). Figure 3.5 shows the Bytes/Flop results for the Sandia and SPEC floating point suites. While the Sandia suite shows a maximum of a 66% increase over the SPEC FP suite, at level 1 cache sizes the difference is between 28% and 35%. At very large cache sizes, the Sandia suite performs 25% more memory work than SPEC FP.

3.7 Spatial Locality Characteristics

This experiment is the orthogonal counterpart of the temporal locality experiment described in 3.6. The objective of this experiment is to understand the characteristics of each application's *spatial working set*, over a fixed interval of instructions. The spatial working set, for a particular interval of instructions, is the total number of unique words that interval references.

The spatial working set is modeled as a infinitely large cache. The same interval of 4 billion instructions as was used in the temporal working set experiment was used to determine the spatial locality. Each unique native machine word requested by the program is recorded in the cache over the course of the instruction stream. This is represented as a bitfield (one gigabit in size) in which every machine word in the virtual address space of the program being profiled is represented by a single bit.

In real memory systems, memory is fetched not in units of a single byte or even a single machine word, but in blocks, so useful insight can be gained by breaking this bit vector into blocks and looking at not only how many blocks are required but at what percent of these blocks are never touched.

At the end of the experiment, the required words are clustered into block sizes from 4-bytes (a single native word) to 8 KB (a typical small page). For every block used, the number of untouched bytes in that interval is recorded.

The spatial working set is understood using three key measures:

1. The unique bandwidth requirement for the instruction stream: specifically, the **minimum** number of blocks (of varying sizes) required to fulfill the interval's data needs. This provides a lower bound for the amount of bandwidth required from the memory system.
2. The overhead represented in the bandwidth given in (1): specifically, this is the number of untouched bytes in each clustered cache block divided by the block size.

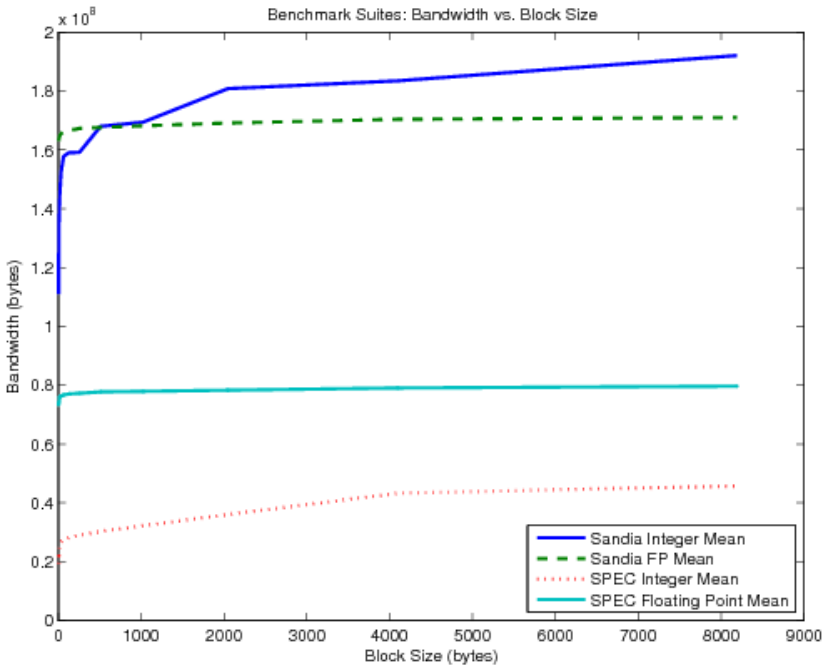


Figure 3.6. Mean Benchmark Spatial Locality Results

3. The total number of unique bytes used by the instruction stream. (This is independent of block size.)

Where the temporal working set is, effectively, a measure of how far back in time a given word was accessed, the spatial working set is a measure of the number of unique words used over a period of instructions.

Figure 3.6 shows the spatial locality results for each benchmark suite (individual results are in Appendix A, Figure A.4). The results represent the minimum required bandwidth for a given cache block size, starting with a single word and progressing through an 8 KB page. The Sandia Floating Point benchmark suite requires between 2.15 and 2.22 **times** the unique bandwidth of SPEC FP. The Sandia integer suite consumes between 4.20 and 5.72 **times** the unique bandwidth of its the SPEC Integer counterpart. The single-word block size results demonstrate the significant difference in the size of the data sets required by the Sandia codes, as opposed to the

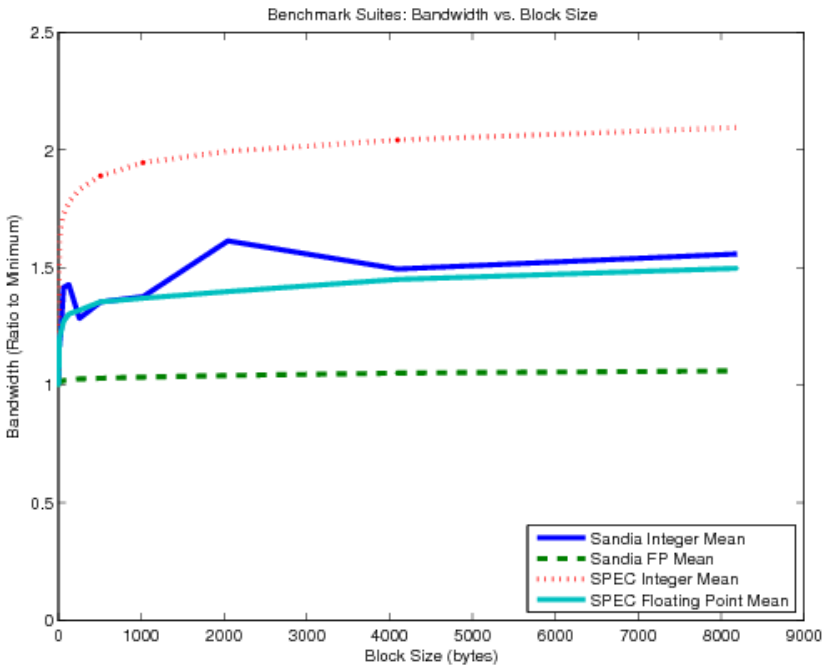


Figure 3.7. Mean Benchmark Spatial Locality Overhead Results

SPEC codes. This is because each word is entirely consumed by the calculation. As the block sizes increase, the "unused overhead" in reading larger blocks is exposed (since the program used precisely the single word spatial working set and incurred overhead by increasing the block size).

Finally, the overhead (in terms of unused data read in while the blocksize increases), is shown in Figure 3.7 (full results in Appendix A, Figure A.5). The absolute minimum bandwidth is given as the first point on each curve in Figure 3.6. Specifically, reading in all the data with a blocksize of 4 bytes is minimal. Any increase in blocksize may waste space. Consequently, Figure 3.7 shows the *additional* bandwidth required as a ratio to the minimal bandwidth requirement by the following formula:

$$O_x = \frac{B_x}{B_{min}} \tag{3.1}$$

Where O_x is the overhead of blocksize x, B_x is the experimentally measured bandwidth required for blocksize x, and B_{min} is the minimum required bandwidth (specifically, the bandwidth for a blocksize of one word). The Sandia Integer benchmark suite has the only non-smooth shape, at small blocksizes and at blocksizes around 2 KB, which is due entirely to the BLAST showing very poor results at these block sizes.

3.8 Conclusions

Both the Sandia integer and floating point suites demonstrate significantly more challenging memory requirements than their SPEC counterparts. Most strikingly, the Sandia floating point benchmarks perform over 5.5 times the number of integer operations per flop of their SPEC FP counterparts. In terms of the temporal working set, the Sandia applications show larger, more irregular strides of their SPEC counter-parts, with a potentially 70% to 80% degradation in performance as a result. The spatial locality results show that the Sandia floating point benchmarks require over 2 times the unique bandwidth of their SPEC FP counterparts, and the integer benchmarks require over 4 times the unique bandwidth of the SPEC integer suite. This demonstrates that the Sandia applications consume significantly more unique data, despite any cache configuration that may capture that data, and that it is significantly harder to manage that data temporally.

CHAPTER 4

THE TRAVELING THREAD EXECUTION MODEL

The *execution model* for an architecture describes the critical features of the architecture to the programmer or compiler writer. For example, in the CC-NUMA execution model, a load will guarantee that a "current" copy of the requested data will be made available to the processor requesting the memory item. Similarly, a store provides for the orderly update of a memory location according to the coherency protocol. The *Traveling Thread execution model* fundamentally represents a memory-centric view of computing, as opposed to the current processor-centric model. In today's machines, processors are explicitly named resources upon which threads (or processes) execute. Canonically, a memory reference by the processor results in a particular "dumb" memory location being read or written. In an architecture with hierarchical memory, that memory reference must negotiate each level of the hierarchy between its home location and the processor. In a uniprocessor system, if the reference is truly unique, the memory object resides in DRAM and is brought through however many levels of cache the processor has, and, in RISC machines, placed into a register for modification. In fact the register is yet another level of memory hierarchy, but instead of being an *implicit* structure, like a L1 or L2 cache in a conventional processor, its contents are *explicitly* managed by the programmer or compiler. That is, the implementation choices expressed by a given cache hierarchy are typically architectural details that are purposefully obscured

from the programmer's view. This is in contract to the PIM environment where on-chip memory is treated as a part of the address space.

There are also software controlled caches, which are popular in embedded systems and are explicitly exposed to the programmer. These caches tend to be used more like overlays in that segments of data are explicitly brought into the smaller/faster area of the memory hierarchy by the programmer, modified, then explicitly written out so that a new data segment can be brought in. In most conventional machines, the term "cache" refers to an implicit structure where memory references are issued by the programmer and the hardware manages all the state involved in holding copies of the data in the cache's faster memory structure. In a multiprocessor environment, particularly with shared, cache-coherent memory, the machinations resulting in a memory read or write are even more complex, with the coherence protocol dictating how copies of a given memory location are shared.

On the other hand, the memory-centric view allows the computation to execute on whatever processing resource is available near the memory location requested. That is, the computation follows the memory, rather than being tightly bound to a particular processor. This is particularly advantageous in a PIM environment for several reasons:

1. the processors are simpler than conventional out-of-order machines, they are more plentiful, and can be interspersed with memory inexpensively;
2. the processors and memory reside on the same chip, memory latencies (for local accesses, at least) are significantly shorter than in a conventional machine; and
3. in a PIM supercomputer, the objective is to scale the amount of memory (and the number of processors going with it) to extremely large sizes, where coherency protocols simply cannot function.

The third point is of particular importance. Today's cluster-based supercomputers tend to sacrifice any advantages of a shared address space for scalability up to tens of thousands of nodes, simply because the coherency protocols impart too much

overhead at that size (they work only up to hundreds of nodes). In the case of a PIM-based system, where potentially millions of nodes are proposed in a peta-scale design, the typical programming model for supercomputers of 10's of thousands of nodes (e.g., MPI) may not scale. That is, an MPI (or OpenMP, or UPC) program requires significant intervention on the part of the programmer to scale the number of processors by 2-4 orders of magnitude. The Traveling Thread model provides the advantages of a shared memory space, without the overwhelming cost of a coherency protocol. Furthermore, the model exposes numerous opportunities for the compiler (and the programmer) to find concurrency in small threads. Existing programming models can be leveraged to provide some parallelism (perhaps 10 thousand-way), and traveling threads can be used below those models to expose more.

The key question is whether or not such opportunities can be utilized.

4.1 The Implications of Prior Work

This work directly results from the exploration of traditional data placement and execution models for very large PIM arrays. The results generated by prior work are extensive [77, 79, 78, 66]. There are two critical contributions of the prior research in this area: first, extensive benchmarking of data intensive applications has provided insight into the structure of these benchmarks and their mapping onto PIM; second, an initial implementation of the traveling thread model using a *carpetbag cache* [78] proved beneficial for a smaller set of data intensive benchmarks. This work significantly extends the original of benchmarks, which has, over time, proven somewhat small for supercomputer-scale benchmarks. Furthermore, the original carpetbag cache implementation was designed as a mechanism to capture the state of relatively large threads. This work significantly improves on that attempt by identifying the potential concurrency available in very small threads, and by free-

ing the instruction streams under study from artifacts of how those streams were compiled. For example, Chapter 5 studies true dataflow graphs in the instruction stream, focusing on the creation and consumption of intermediate results, rather than analyzing how those results happened to be represented in registers and stack memory for a given target architecture.

In contrast to prior work, this work has a two-fold increase in focus: first, to demonstrate that there is significant room for performance improvement in existing applications, and that that improvement can be captured by the traveling thread execution model, particularly as it applies to large, distributed PIM arrays; and second, to identify the types of concurrency that compiler writers should develop algorithms to extract in order to fully take advantage of this execution model. Consequently, the execution model presented here offers the architect a range of implementation choices for thread creation and termination, synchronization, and migration. This is done purposefully, as the "most correct" choices will largely depend on how successful compiler writers are at extracting the type of parallelism described in this dissertation.

4.1.1 Prior Benchmarks

Prior benchmarking focused upon the Data Intensive Systems (DIS) suite [12], the SPEC 95 suite [83], a Molecular Dynamics Simulation [46], and object oriented databases [29]. The SPEC suite represents a collection of typical workstation workloads, and is designed to be easily captured in a cache. Consequently, PIM (which has a much larger on-chip memory space) can easily capture most of the dataset. The DIS suite, on the other hand, represents a collection of code intended to exhibit low reuse and requires more extensive data movement. This type of program executes poorly on conventional architectures, but executes relatively well on PIM

systems. Supporting data intensive operations is a critical measure of distinction between PIM (and the traveling thread execution model) and conventional systems. Next, the molecular dynamics simulation represents a data and computation intensive benchmark, for which petaflop-scale computation is targeted. In fact, the IBM BlueGene/C effort[5, 108] addresses only this application. Finally, the object oriented database proved enlightening compared to a standard relational database, because it introduced more indirection as object abstractions were represented in memory.

While the DIS suite proved significantly more "interesting" (from the perspective of the memory system) than the SPEC 95 suite, it was ultimately a "benchmark" suite, and provided only simplified inputs. One of the main contributions of this work was to identify a suite of supercomputing applications and quantify precisely how their memory characteristics are different from SPEC. Consequently, the benchmarks used in this work are significantly larger and more representative than any used in prior work.

4.1.2 The Original Carpetbag Cache

Figure 4.1 depicts a *Carpetbag Cache*, which is a small, mobile cache that can be associated with a thread and facilitates its traveling by capturing frequently used data. Prior experimentation indicates that when a traveling thread moves from its *source node* (where it is executing when a remote reference is generated) to the *target node* (or, the location of the remote data), taking a small amount of data from the source node to the target helps avoid thrashing between two nodes, and generally increases the thread's runtime on the target. More generally, however, a carpetbag cache can be thought of as capturing data on numerous preceding nodes of computation for continued use.

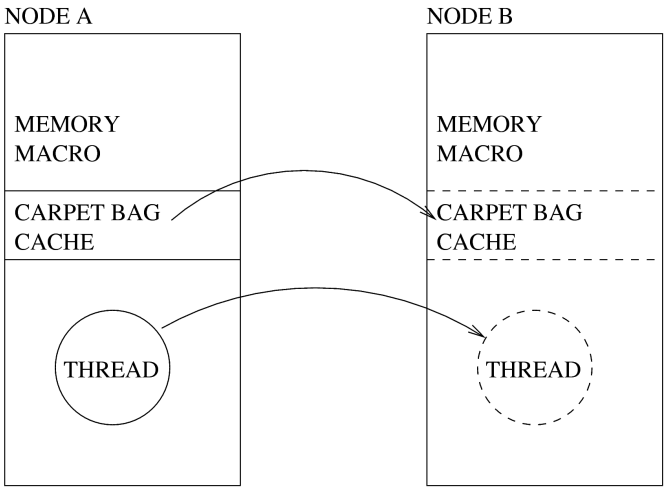


Figure 4.1. Carpetbag Cache Example

The explored caching and synchronization models have, thus far, been very primitive. The threads only retain data from the node where they executed previously. That data is kept coherent by means of *full/empty* (see Section 2.5.2) bits for every word in memory. When a word is placed into the carpetbag cache it is *checked out* and cannot be accessed until it is released by the thread that checked it out. This is because that thread adhered to the rule that upon moving, only data from the prior node of execution is retained. Consequently, it flushed its previous carpetbag cache back to the owner, in favor of the new cache constructed on the current node.

This mechanism requires enhancement as part of the experimentation for this work.

Finally, it should be noted that all prior work concentrated on relatively **large** threads. That is, the amount of bandwidth required to transmit the thread's state was much larger than in this work, precisely because the runtime of the thread was significantly longer. In attempting to exploit opportunities for more concurrency, this work focuses on much smaller threads, and consequently much smaller thread

state sizes. The problem is, of course, that very few machines support short thread lengths (the MTA being the best example). Even in the MTA, the thread **state** is very large; in excess of 64 64-bit registers! Consequently, no compiler currently in existence compiles for threads of such small state.

4.1.3 Relevance to the Current Work

The prior work in the area of traveling threads indicates that there is a demand for a simplified execution model capable of reducing the overhead required to access remote data. The traveling thread model converts round-trip remote accesses into one-way accesses, and has been demonstrated to increase a thread's execution time between remote accesses on a given node[78]. While PIMs are clearly capable of sustaining extended computation when configured as "traditional" SMP or CC-NUMA systems, extending those models to extreme (trans-petaflop) scale systems is very costly. This is true in terms of both the overhead of name resolution for remote memory (ie, maintaining page tables and cache coherency state) as well as the overhead of the remote access itself.

4.2 The Execution Model

The most basic traveling thread execution model replaces a remote memory access with a thread migration, and is targeted to any PIM-like non-coherent, distributed shared memory environment. To increase compiler-identified concurrency, in support of light weight latency toleration, and reduce the amount of each thread's state, the program is assumed to consist of many, many small threads. In a conventional out-of-order processor, these "threads" would be identified by the hardware. For example, allowing multiple outstanding loads is merely a hardware defined mechanism of allowing several small threads to be launched from a relatively large thread, and the synchronization occurs because the hardware keeps track of which registers

Pseudo-code	**Pseudo-assembly**
`found = 0;`	`add $found, $0, $0`
`listptr = list_head;`	`add $listptr, $list_head, $0`
`while((!found)&&(listptr!=NULL)){`	`loop: bnz $found, done`
	`bz $listpt, done`
`if(listptr->data == search)`	`addi $tmpptr, $listptr, data_off`
	`load $tmp, 0($tmpptr)`
	`sub $tmp, $tmp, $search`
	`bnz next`
`found = 1;`	`addi $found, $0, 1`
`listptr = listptr->next;`	`next: addi $tmpptr,$listptr, next_off`
	`load $listptr, 0($tmpptr)`
	`j loop`
`}`	`done:`

Figure 4.2. This example traverses a linked list. The pseudo-code is given on the left, with pseudo-assembly on the right. The assembly assumes MIPS-like instructions, with registers given names (prefaced with $) for clarity.

still lack the value requested from memory.

4.2.1 An Example Thread

To better elucidate the traveling thread execution model, consider the example thread given in Figure 4.2, which traverses a linked list looking for a particular member. This is a particularly good traveling thread example. In a caching architecture, if the list is large, distributed, and not already in the cache, every access to the current list node (consisting of `data` and `next`) will cause a remote read (unless, of course, two similar structures share the same cache line, which is unlikely for a large, distributed structure). The first access to `listptr->data` will, if the cache line size is large enough, likely pull into the cache both `data` and `next`, but, with

a large list, those elements will not be used much. In fact, the simple check and the loading of the `next` pointer are the only two uses. If the list is shared in a parallel environment, and any of the list nodes reside in other processor's caches, significant coherency overhead may be involved in performing the read (depending on how often `data` is written).

In a traveling thread environment, each access to `listptr->data` may result (either implicitly or explicitly) in a migration of the thread to the node which contains the data to which pointer points (Section 4.2.2 discusses the types of migration). The state required by the thread is very small: the current `listptr`, and the values of `found` and `searct`. Thus the thread can migrate with those three values, whatever internal thread state is required (the thread's program counter, etc.), and potentially a very small number of instructions (which could travel with the thread or simply be distributed to or cached on the relevant nodes).

This example is somewhat simplistic, and, given the large number of Fortran codes in the benchmark suite it is somewhat unrealistic (Fortran has no pointers). However, examples on distributed arrays and other data structures are no different. The difficult questions are: what kind of state needs to be captured? And how much?

4.2.2 When to migrate: the Implications of Fork and Join

In the example thread, there are two data references: first to `listptr->data`, and then to `listptr->next`. These references correspond to the `load` instructions in the pseudo-assembly, and the rest of the variables are considered thread state because they are held in registers. That is, the linked list is the only data structure being accessed in memory. In some respects, this is an artifact of the hand-coded example, but a compiler (particularly with programmer hints) knows quite a bit

about the example code. For example, the compiler knows that both `data` and `next` are part of the same linked list data structure (it has to know their offsets to be able to perform accesses to the structure). Given very primitive rules about memory allocation (such as each component of a dynamically allocated data structure must reside on a single PIM node), the compiler could know that both `data` and `next` are co-resident. Given a big enough register state, the compiler also has likely made the same register allocation decisions as in the example. Both `found` and `listptr` are loop control variables, and consequently most likely placed in registers.

The only point at which migration may occur in the example, and likely the way a compiler would generate code, is during the test of `listptr->data`, and architecturally several implementations are possible within the same traveling execution model:

1. The migration could be entirely handled by the hardware. In fact, the original carpetbag cache experiments worked in exactly this fashion when a memory reference was determined to be non-local. The thread state, consisting of its register set and the contents of its carpetbag cache, was packaged and then moved to the remote node.
2. The migration could be handled entirely by software. The software could test whether or not the current `listptr` was local, and if not it could explicitly migrate.
3. A combination of hardware and software mechanisms could be used via an exception. The thread could trip an exception (e.g., the "non-local address access exception") by accessing the remote pointer, and control could be passed back to the software to decide what to do about it.

For the purposes of the execution model in this dissertation, the actual mechanism is immaterial. What is critical is identifying the state required to perform the migration. In fact, what is meant by "migrate" can also have multiple meanings:

1. The thread in question could launch a thread on the remote processor containing the data pointed at by `listptr` explicitly, and either terminate the existing thread or wait for the answer;
2. the existing thread itself could move, taking its thread state with it; or

3. the existing thread could, in fact, issue a remote load requesting the data.

The implications of the choice above are, again, less important to this work than understanding when threads should migrate, and what kind of state is required during that migration.

4.2.3 Naming and Name Resolution

This simple example has brought out a critical piece of the execution model. It must be possible to quickly determine which addresses are local and which are non-local, and there must be a mechanism for resolving where non-local addresses reside. The former requirement is, of course, intuitive because no local load or store could execute if the node performing that memory operation did not know where in its local memory that load or store was to reside. The latter (address resolution) requirement is a complex question, but the ability to determine a remote memory address' location must also logically be available to the thread in any shared memory machine. For the purposes of this work, each PIM receives a single contiguous, non-moving chunk of the address space, making local/non-local determinations and decisions about routing during migration simple. In the case of a software migration, this knowledge must be exposed to the programmer (e.g., via some mechanism to test if an address is local). Assume the instruction `testlocal $address, label` jumps to `label` if the address is local, and the instruction `migrate $address` migrates to the node containing `$address`.

The pseudo-assembly for a software controlled migration is shown in Figure 4.3.

The exception mechanism for a non-local test looks similar, except that the exception handler performs the migration instead of the thread itself, then restarts the load once it is known to be at the correct node. In this case, the exception handling code starts at the label `nonlocal_exception`. During thread start-up,

```
      add $found, $0, $0
      add $listptr, $list_head, $0

loop: bnz $found, done ; while
      bz $listpt, done

      testlocal $listptr, local ; if $listptr is nonlocal then migrate
      migrate $listptr

local:
      addi $tmpptr, $listptr, data_off ; if
      load $tmp, 0($tmpptr)
      sub $tmp, $tmp, $search
      bnz next
      addi $found, $0, 1 ; found=1
next: addi $tmpptr,$listptr, next_off ; listptr = listptr->next
      load $listptr, 0($tmpptr)
      j loop
done:
```

Figure 4.3. Explicit Thread Migration

the exception handler is registered with the system (in the same way an interrupt handler would be registered). The register `$resume` contains the program counter where the exception occurred, and `$exception_address` contains the address that caused the exception. When the non-local load occurs, the processor invoked the exception handler (supplying it with `$exception_address` and `$resume`), which performs an explicit migration, then resumes the load once the thread has reached the node that owns the address that was previously non-local.

The pseudo assembly is shown in Figure 4.4.

If the migration were to occur automatically, the hardware would perform the same test, but the loop could be executed as originally written in Figure 4.2.

```
      add $found, $0, $0
      add $listptr, $list_head, $0

loop: bnz $found, done ; while
      bz $listpt, done

      addi $tmpptr, $listptr, data_off ; if
      load $tmp, 0($tmpptr) ; the exception may occur here
      sub $tmp, $tmp, $search
      bnz next
      addi $found, $0, 1 ; found=1
next: addi $tmpptr,$listptr, next_off ; listptr = listptr->next
      load $listptr, 0($tmpptr)
      j loop
done:
      .
      .
      .
nonlocal_exception:
      migrate $exception_address
      jr $resume
```

Figure 4.4. Migration via an exception

4.2.4 Thread State

The critical question, with respect to migration, is what type of thread state should be migrated? The original traveling thread work, which focused on a carpetbag cache, assumed that the state to be migrated was a combination of the thread's register set and some number of recent references from the current node of execution. In fact, a small carpetbag cache been shown to effectively capture the state for large threads, and significantly reduce the number of remote migrations. However, the thread given in the above example is a small thread (it consists of perhaps a dozen instructions, two permanent registers, and two temporary registers). If the list in question is acyclic, then, in fact much of the carpetbag cache would be of

very little use to this thread. That is not to say that other threads would not benefit from taking more local information with them, as this example is purposefully simple. However, in this case, the core state could be represented by four registers: a thread status word (including the program counter), and the registers `$searct`, `$found` and `$listptr`.

4.2.5 Synchronization

In the now familiar linked list example, no synchronization was given. Any thread that returns a value will have to perform synchronization with the thread requiring that value (via either a data synchronization, in the form of locking, or a control synchronization in the form of a join which indicates that the thread has completed its work). In a highly concurrent environment, additional synchronization may be required. For example, if, in addition to the list search thread, the list were to be updated, then the values of the global data structure pointed at by `listptr` would have to allow some form of atomic operations to occur. In terms of the traveling thread execution model, the only requirement is that some sort of lightweight synchronization mechanism exist. The following basic functions must be provided:

1. Read the value for the given word if it is not being updated;
2. Write the value for the given word if it is not being updated;
3. Lock the given word so that an update may begin;
4. Unlock the given word at the end of an update.

In the original traveling threads experiments, this was implemented via a set of full/empty bit primitives, associated with every word in memory. These primitives existed on the MTA as well as the HEP (see Section 2.5.2), and were described in detail in Section 2.5.2. However, this is not the only lightweight synchronization

mechanism available. A lightweight version of the PowerPC's memory reservation mechanism would meet these requirements: the Load Word and Reserve Index (`lwarx`) and Store Word Conditional Index (`stwcx`) instructions work as follows:

- `lwarx Rt, offset, Rs`: loads the value at memory address $Rs + offset$ (where Rs is a source register and offset is an immediate integer value) into register Rt, and place a *reservation* on the address $Rs + offset$, if no such reservation exists already. Any store instruction will clear the reservation, allowing other waiting loads to obtain it.
- `stwcx Rs, offset, Rt`: stores the value of Rs into memory location $Rt + offset$, if the address $Rt + offset$ has previously been successfully reserved by the `lwarx` instruction. If the store completes successfully, a condition register is set to true, otherwise no update is made and the condition register is set to false (for later inspection).

The PowerPC architecture limits the number of outstanding reservations in most implementations to one, which would most likely be too few allowable synchronizations for small threads (See Chapter 6).

Similarly, the UltraSPARC architecture provides a "test and swap" (`swap`) instruction which performs all of these functions as its primary lightweight synchronization primitive, with a Register Transfer Language (RTL) description similar to the following:

```
swap $testaddr, $swapreg:
   lock($testaddr)
   if(memory($testaddr) == 0) {
     memory($testaddr) = $swapreg
     $swapreg = 0
   }
   unlock($testaddr)
```

This is more heavyweight than the fine-grained control offered by full/empty bits. However it does perform all the basic functions. Loads and stores to `$testaddr` cannot occur in the middle of a `swap` instruction. The `$swapreg` is meant to contain the thread ID for the lock (which is given by `$testaddr`), and the value 0 is written in when the lock is free. The spatial overhead is relatively high (an entire machine word

```
; increment listptr->data

addi $tmpptr, $listptr, data_off  ; compute the address
lock $tmpptr                      ; lock $tmpptr
load $tmp, 0($tmpptr)             ; load the value
addi $tmp, $tmp, 1                ; perform the increment
store $tmpptr, $tmp               ; store the result
unlock $tmpptr                    ; unlock the address
```

Figure 4.5. A writer incrementing the data value of a node in the linked list.

for every lock that is required), and very few swap instructions can be outstanding at any given time. However, it is optimized for an architecture with relatively few threads and only coarse-grained, programmer-controlled locking.

For the purposes of this work, the implementation of the locking scheme is less important than identifying where the synchronizations occur. Consequently, synchronization is performed using the following primitives:

1. **load:** loads the value from memory if it is not currently locked by another thread, otherwise blocks until it is unlocked;
2. **store:** stores the value into memory if it is not currently locked by another thread, otherwise blocks until it is unlocked;
3. **lock:** locks the given memory address if it is not locked by another thread, preventing loads and stores from other threads, otherwise blocks until it is unlocked;
4. **unlock:** releases the lock on the given memory address.

Figure 4.5 shows the update of `$tmpptr` using the explicit synchronization provided for above. Semantically, the load and the lock could be combined (as it is in the MTA), or it could be multiple instructions as given above. In fact, the entire operation could be made into an atomic increment instruction. Chapter 6 discusses the implications of explicit synchronization (as given above) versus implicit synchronization (which happen during every memory updating instruction) in detail.

Synchronization is not limited to main memory words: registers shared between threads would require the same synchronization. It is similarly implicitly performed in a CC-NUMA environment on cache lines.

4.3 Conclusions

This chapter described the fundamental traveling thread execution model, as well as the required generic architectural support. In general, the model is applicable to a number of distributed shared memory architectures, with multiple mechanisms for migration, thread creation and termination, and synchronization. For the purposes of this work, identifying the *types* of events that occur (thread creation, destruction, thread migration, and required synchronizations) and their frequency is more critical than identifying the specific mechanism. Chapter 8 will take the experimental results for each of these areas and identify an appropriate design point. Generic mechanisms identified in this chapter, and used in the subsequent experimentation, allow for the evaluation of multiple potential implementations. For example, by establishing the requirements for synchronization, and the number of synchronization events for various thread sizes, Chapter 6 will show for any thread length and average synchronization overhead, the expected performance penalty paid in synchronization events.

CHAPTER 5

DATAFLOW

This chapter provides a detailed analysis of the *dataflow graphs* given by the benchmarks under study. A *dataflow graph*, discussed in greater detail in Section 5.1, represents the computation performed by the application as a series of operations and the data required to perform them. In this way, specific architectural artifacts, such as how the programmer or compiler chose to allocate registers, are removed, allowing for an independent analysis of the computation. This is the first step in understanding how best to identify traveling threads, and what their data requirements are.

In addition to the specific computer architecture implementations discussed in Chapter 2, which all benefited from extensive application profiling to choose the most relevant implementation features, there is an extensive literature on Dataflow Graphs and how they can be scheduled [85, 21, 58].

The remainder of this chapter is organized as follows: Section 5.1 discusses dataflow graphs, how they can be scheduled in serial and in parallel, and the critical measures presented in this chapter; Section 5.2 shows the results in terms of input and output to the graph, available concurrency, minimum latency, data sharing, and the lifetime of that data; Finally, Section 5.3 presents the conclusions.

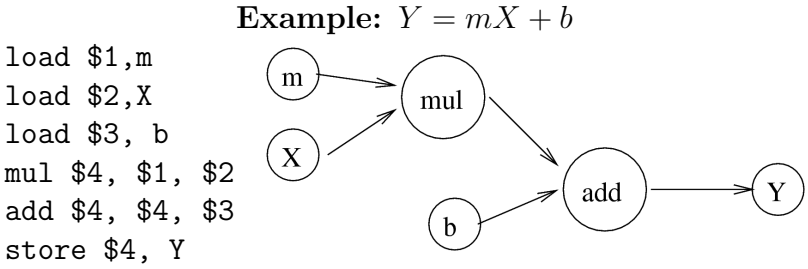


Figure 5.1. The dataflow graph and pseudo-assembly language implementing the computation of $Y = mx + b$

5.1 Dataflow Graphs

A dataflow graph, $G = (V, E)$, is a directed, acyclic graph (DAG) that consists of a set of vertices (V) and edges (E). The vertices represent computation operations to be performed, or, in the case of data items required by the computation and accessed solely by a `load` or `store`, the graph's *inputs* and *outputs*. The directed edges link the vertex that produced a required data item with the vertices that consume that data item.

Figure 5.1 shows an example dataflow graph for computing $Y = mX + b$. In this example, the inputs are represented by the memory items m, X, and b, and are accessed via a `load` instruction. The output, Y is produced by a `store`. Each of the edges represents a required data item. For example, the multiply of m and X is performed through registers `$1` and `$2` (that were loaded in the first and second instructions). This produces the edges: (m, mul) and (X, mul). The other directed edges in the graph are: (mul, add), (B, add), and (add, Y). By constructing this graph, the registers (`$1`, `$2`, `$3`, and `$4`) that the programmer or compiler used to encode the desired computation have been abstracted. In effect, the dataflow graph shows what the computation in a memory-to-memory architecture would look like.

In a memory-to-memory architecture, the program in Figure 5.1 becomes:

```
mul tmp, m, X
add Y, tmp, B
```

The dataflow graph *does not* remove all artifacts of the architecture, only how the data dependencies are encoded. For example, the operations in the dataflow graph are the instructions (`mul`, `add`, etc.), supported by the ISA. The ISA under study is generally a RISC ISA, so, in this case, the dataflow graphs show relatively simple operations. In an ISA with a multiply-accumulate instruction, the encoding would be different (e.g., it would contain fewer vertices representing instructions).

The construction of a dataflow graph from the instruction stream is space intensive because the dataflow graphs keep track of (and eliminate) memory dependencies. For example, the following sequence of instructions increments the memory value X and later uses loads the incremented value for some other use:

```
load $1, X
addi $1, $1, 1
store X, $1
.
.
.
load $1, X
```

The value produced by the `addi` instruction is stored into memory location X, and later consumed again (via the load instruction). In terms of the dataflow graph, that value was produced by the `addi` instruction, which was the last instruction to modify the value (even though the store wrote the value to memory).

Figure 5.2 shows the dataflow graph produced by the example code. The vertex that last produced a value (in this case the, `addi` instruction) is used as the source vertex of that value, regardless of how that value passed through memory or registers. Consequently, for every register or memory location consumed by the

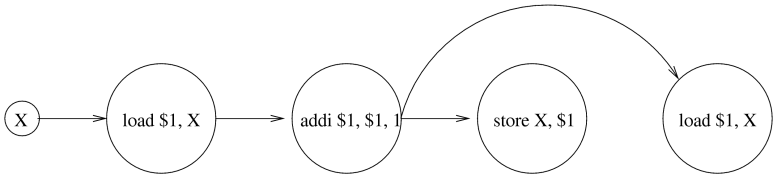


Figure 5.2. Dataflow Dependencies Passing Through Memory

instruction stream, the instruction which produced the value that the register or memory location holds must be tracked (so that if the location is loaded and used again, an edge in the graph can be created). The dataflow graphs used in this chapter are limited to $1,000,000$ instructions (the number of vertices varies because of input and output vertices). Because the goal of this dissertation is to focus on short threads (e.g., tens of instructions), this architecturally relatively small $1,000,000$ vertex graph is huge by comparison to the projected size of a thread.

5.1.1 Dataflow Graph Construction

Dataflow graphs are constructed from an instruction stream by a simple algorithm that tracks which instructions produced every register and memory value for a given instruction stream. The algorithm processes each instruction in program execution order as follows:

1. Create a memory map that takes a memory address a and returns the vertex that produced the value in a.
2. Create a register map that takes a floating point or integer data register, and returns the vertex that produced it.
3. For each instruction in the instruction stream (in the given execution order):
   - Create a vertex, v, to represent the instruction.
   - For each register that the instruction v reads, use the register map to determine which vertex (u) produced that data value. If the map does not contain the vertex u for the given register, the vertex is created and inserted into the map (because the data item is an input). Create the edge (u, v).

- For each register that the instruction v writes, update the register map to indicate that v produced that register.
- If the instruction is a load, consult the memory map to determine which vertex (u) created the addressed data item. If no such vertex exists, create it to represent the input. Update the register map to indicate that the register being loaded into was created by vertex u.
- If the instruction is a store, use the register map to determine which vertex created the register to be stored (u). Update the memory map to indicate that the address being stored to was produced by the vertex u.

This algorithm produces edges in the dataflow graph based on the instruction that produced the data value, regardless of how that value passed through registers or memory. There are several interesting properties:

- Load and Store instructions do not actually produce data values (they move them to and from memory), however they do consume data values during the computation of an address. For example, `load $1, 0($2)` uses the data value in register `$2` to compute the address of the data to be loaded into register `$1`. Consequently, it is represented in the dataflow graph as a vertex that consumes one data value and produces none (since register `$1` was actually produced by another instruction before it was stored to memory).
- Conditional Branches do not produce any programmer visible registers. They conditionally alter the program counter, affecting the control of the program, and they may consume programmer visible registers (e.g., in testing for a condition). Consequently, they also appear in the dataflow graph as instructions that consume values but do not produce them.
- Immediate values (e.g., small integers encoded within instructions rather than represented in registers or in memory) do not appear in the dataflow graph. The instruction `addi $1, $2, 4`, consumes the register `$2`, and produces the register `$1`, but the constant 4 is unaccounted for.

Although these particular vertices may appear unusual, they maintain two critical properties of the dataflow graph: first, the dataflow graph accounts for every use (e.g., consumption) of a data item; and second, all values consumed are represented by the instruction that produced those values, not by how those values happen to pass through memory or registers.

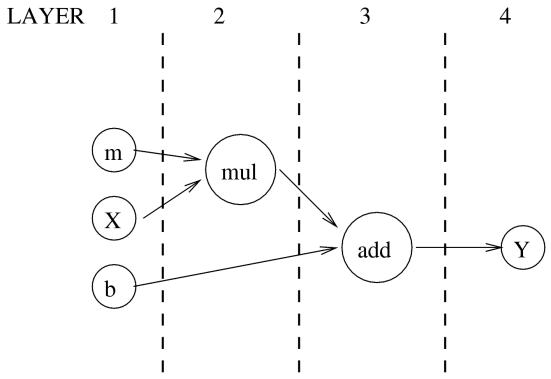


Figure 5.3. The dataflow graph from Figure 5.1 assigned a schedule that allows instructions to execute only after each of their data dependencies are satisfied.

5.1.2 Scheduling

Dataflow graphs establish a data dependent ordering of operations. That is, in the example graph representing $Y = mX + b$, the multiply operation cannot occur before m and X are loaded from memory, and the add cannot occur before b is loaded and the results of the multiply instruction are available. In an architecture capable of dispatching single-instruction threads, the schedule would be determined by the dataflow graph. In fact, methods of Out-of-Order execution, such as Tomasulo's algorithm[104] or scoreboarding[51], execute instructions in compliance with the dataflow graph (register renaming, in fact, removes the exact register encoding dependency discussed in Section 5.1, and has been discussed extensively in the literature[51]. In order to perform an architectural analysis of dataflow graphs, each operation in the graph must be given a *schedule*, or assigned an execution time that occurs after all of the vertices data dependencies are satisfied.

Figure 5.3 shows precisely such a graph for the original example given in Figure 5.1. The schedule is constructed via *topological layering*, which is described in detail in Section 5.1.3. Intuitively, however, instructions can only be executed once the

data that they require has been computed. The example program consists of 6 instructions, but the dataflow graph shows that several of those instructions can be executed in parallel. The schedule graph shows that the three loads can be executed in *layer 1* (or at timestep 1), followed by the multiply at *layer 2*, then the add at *layer 3*, then finally the store at *layer 4*. This analysis assumes that all instructions execute in a single timestep, which is not true on modern architectures. For example, floating point divides take multiple cycles. A dataflow graph can be constructed to more accurately reflect timing by inserting additional vertices for multiple timestep operations, or by weighting the edges. However, this would require fixing the details of the architecture in question. In this case, single-timestep operations most accurately show the data dependencies. An architecture capable of identifying all the data dependencies in the dataflow graph, and of executing an infinite number of instruction could execute the example computation in 4 timesteps.

There are several critical properties given by the layered topological ordering:

- Each vertex's **indegree** (d_{in}), or the number of edges that terminate at the vertex, shows the number of arguments consumed by the data operation. For example, the `mul` vertex of the graph has an indegree of 2, from m and X. In general, this is a feature of the architecture, which in this case limits the number of registers consumed by the architecture to 3. The maximum indegree for any vertex in the graph is therefore 3.
- Each vertex's **outdegree** (d_{out}), or the number of edges originating at the vertex, shows the number of copies of data items produced by the instruction. In the case of `mul`, the outdegree is 1. The PowerPC under study limits the number of data items produced to two, however reuse creates an edge from the instruction in question to the instruction consuming the data item every time the data is consumed, making the outdegree potentially much larger than 2.
- The **critical path** through the code segment can be found by counting the number of topological layers, in this case 4, and represents the minimum latency required to perform the computation.
- Each layer's **concurrency** is defined as the number of vertices present in that layer. For example, layer 1 has 3 vertices. Each of those instructions can be executed in parallel, provided that all the prior layer's instructions have been executed.

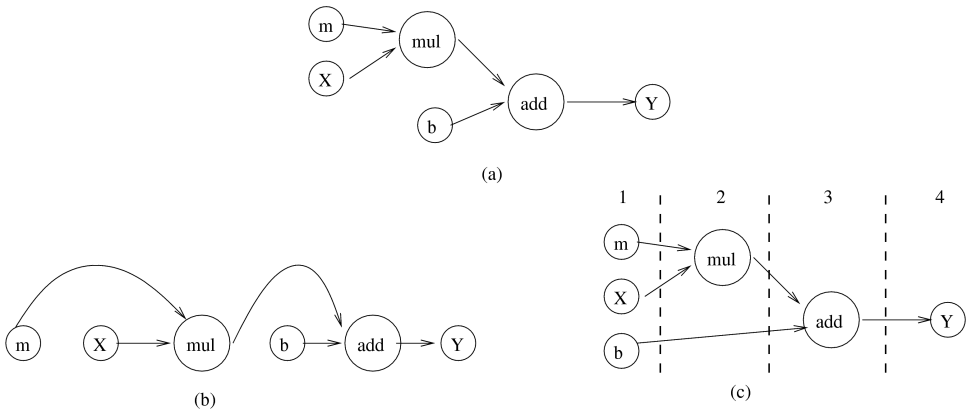


Figure 5.4. The scheduling for the example dataflow graph: (a) The original dataflow graph from Figure 5.1 shown in (b) topological order and (c) layered topological orderer with an ASAP schedule.

- Every edge in the graph has a **use distance** representing the number of timesteps between the time the data item was produced and when it was consumed. For example, m is produced at timestep 1, and consumed at timestep 2, giving it a use distance of 1 timestep. Similarly, b has a use distance of 2 time steps. In typical dataflow graphs, values are reused multiple times, with each reuse being represented by a different edge in the graph.

5.1.3 Topological Layering

The schedule represented in Figure 5.1 is produced by computing the *topological ordering* of the graph, then constructing an As Soon As Possible (ASAP) schedule. A topological ordering of a DAG is a linear ordering of all of its vertices such that, for each edge (u, v) in the graph, the vertex u appears before the vertex v.

Figure 5.4(b) shows the example dataflow graph in a topological order. Multiple topological orders are typically possible, for example, the vertices m and X could exchange topological positions in Figure 5.1(b) to provide a different valid topological ordering. Any correct topological ordering represents a valid schedule for executing the code on an **in-order** processor.

The algorithm for producing the topological layering given in Figure 5.4(c) is as

follows:

1. Topologically sort the DAG and assign each vertex in the graph the topological number (or initial layer) produced by the sort. This produces a valid schedule that has one vertex (or instruction) in each layer.
2. To produce the concurrent schedule, examine each vertex (v_i) of the graph in the topologically sorted order (from left to right):
   - If $d_{in}(v_i) = 0$, the vertex is an input and is placed in the first topological layer
     Else
     - Choose the maximum topological layer (t) from every vertex that has an in-bound edge to v_i.
     - Reassign v_i the topological layer $t + 1$. This schedules vertex v_i *after* all of its dependencies have been computed.

Once this is computed, the critical properties described in Section 5.1.2 can be examined.

5.2 Results

This section examines the properties of the dataflow graphs (as defined in Section 5.1) produced for each of the Sandia benchmarks described in Chapter 3. The full results for each individual benchmark, including graphs showing the amount of concurrency available at each topological layer, can be found in Appendix B.

5.2.1 Dataflow Graph Inputs

Vertices in the dataflow graph with an indegree of zero represent memory inputs to the graph (e.g., data values that are read from memory, not produced over the course of the instruction stream). The input vertices were either generated by instructions executed prior to the start of the instruction stream used to construct the dataflow graph, or are program constants loaded into memory when the program began execution. Fundamentally, the inputs represent **all** the data required to perform the computation represented in the dataflow graph. if those inputs were all

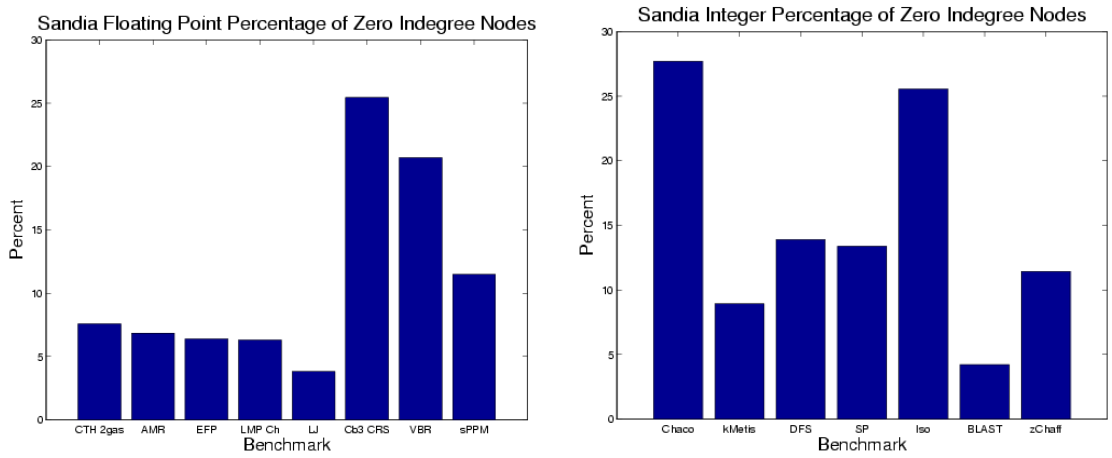


Figure 5.5. Dataflow Inputs and Outputs.

simultaneously available, the dataflow graph could execute its entire computation (e.g., if the computation were implemented as a circuit in hardware, the inputs would be the circuit's *sources*).

Figure 5.5 shows the percentage of zero indegree in each benchmark's layered dataflow graph. The floating point codes that are known to stream through their data (particularly, Cube3) show a very large number of inputs, relative to the amount of intermediate computation that they perform. Similarly, the graph codes in the integer suite (Chaco and Isomorphism, in particular) show a relatively large amount of required input data.

5.2.2 Concurrency

Instructions in the same topological layer can be executed concurrently (e.g., they are guaranteed to be entirely independent of each other). This measure of concurrency significantly affects both how modern superscalar processors can extract parallelism from the instruction streams they execute, as well as the potential for accelerating the application by executing many small parallel threads.

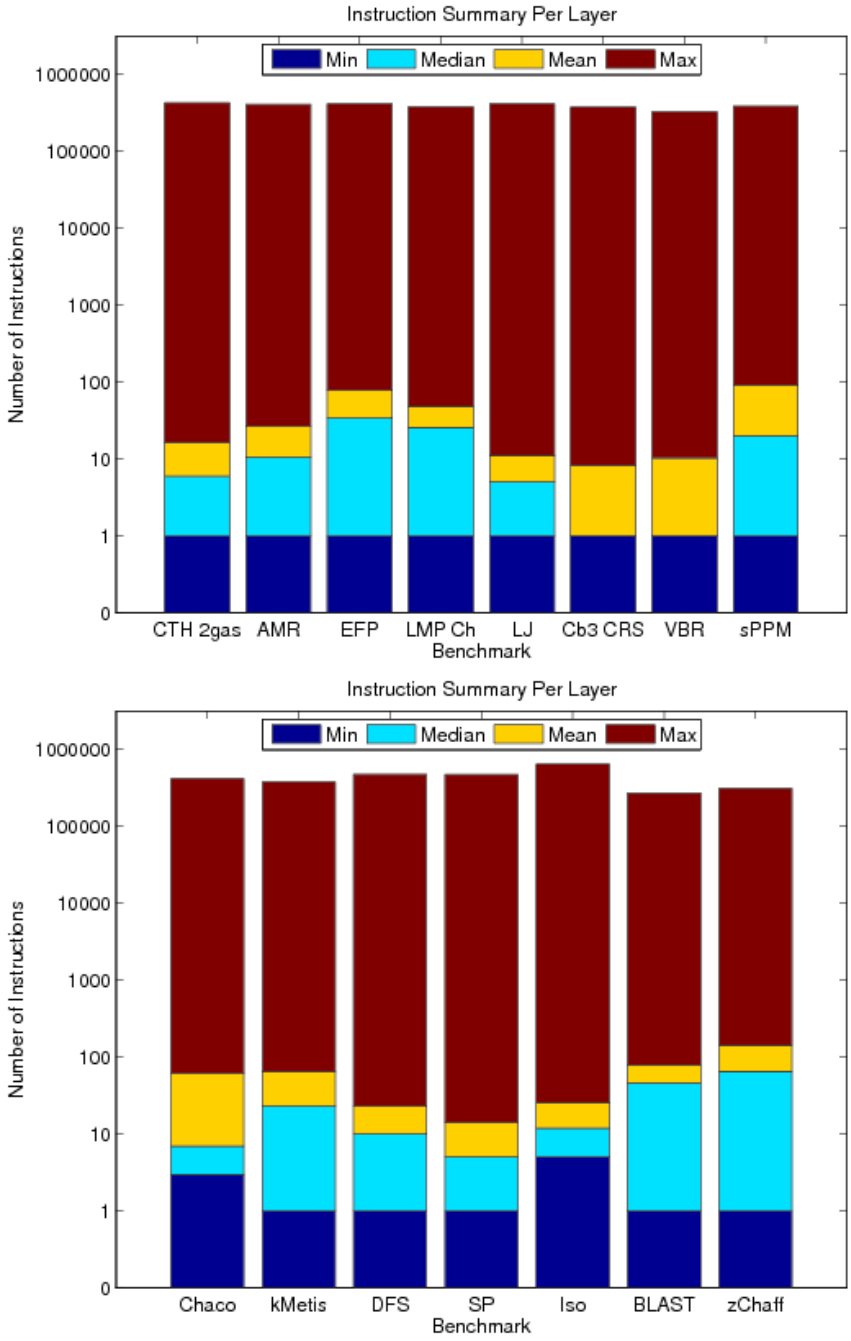


Figure 5.6. Dataflow Graph Concurrency

Figure 5.6 shows the minimum, maximum, median, and mean concurrency available in each one million vertex layered dataflow graph. This was determined by

counting the number of instructions in each topological layer. The maximum is many orders of magnitude larger than the mean and median, which largely represents loads from static addresses which occur in the first layer. The median is presented in addition to the mean because of the size of this discrepancy. Two benchmarks stand out: Cube3 CRS and Cube3 VBR, which have a median of 1 instruction, indicating that they are highly serial codes. This is because the solver performs many dot-products, and the algorithm is written in a serialized data-dependent fashion. Algorithms that expose more parallelism exist, but the dataflow graph is limited to describing the instruction stream's data dependencies (both from memory and registers) as written.

Typically, the median provides tens to hundreds of instructions, with the mean being higher because some sections of code are highly parallel. This demonstrates that modern out-of-order processors, with hundreds of instructions in flight at any given time, may be nearing the end of their ability to tolerate increased memory latency.

Most significantly, the results show that there is potentially a lot of parallelism available in existing instruction streams.

5.2.3 Latency

Figure 5.7 shows the length of the critical path through each of the benchmark's dataflow graph, with the higher bars representing programs with relatively more serial dataflow graphs. CTH 2gas, LAMMPS LJ, both Cube3 runs, the DFS, Shortest Path, and Isomorphism are relatively more serial, with the remaining benchmarks showing, in some cases, nearly an order of magnitude shorter critical path. Interestingly, the "benchmark" CTH run (the 2gas problem) requires a significantly longer critical path than do the production runs (AMR and EFP). The floating point suite

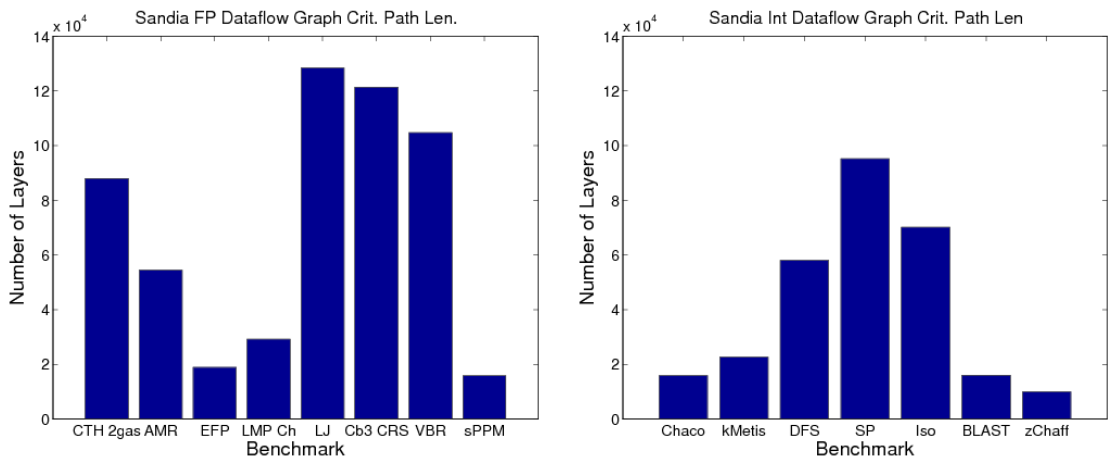


Figure 5.7. Dataflow Graph Critical Path Length

averages a critical path of 70, 118, and the integer suite 41, 149, showing that the floating point suite has a significantly more data-serial computation.

5.2.4 Data Reuse

As discussed in Section 5.1, the indegree and outdegree of a vertex in the dataflow graph describe the data that vertex requires to execute its instruction, and the number of times the results of that instruction are reused respectively. This section discusses the data reuse, as measured by the in- and out-degrees of the vertices. It is critical to note that the mean indegree and the mean outdegree of all the vertices in the dataflow graph will always be the same (by definition). This is because every edge is symmetric. Specifically, an edge (u, v) contributes both to the outdegree of u and the indegree of v. Consequently, this section will discuss both the means and medians. The mean values provide an aggregate description of how many times the results of a particular instruction are reused, whereas the medians describe the variation between the incoming and outgoing edges in the graph.

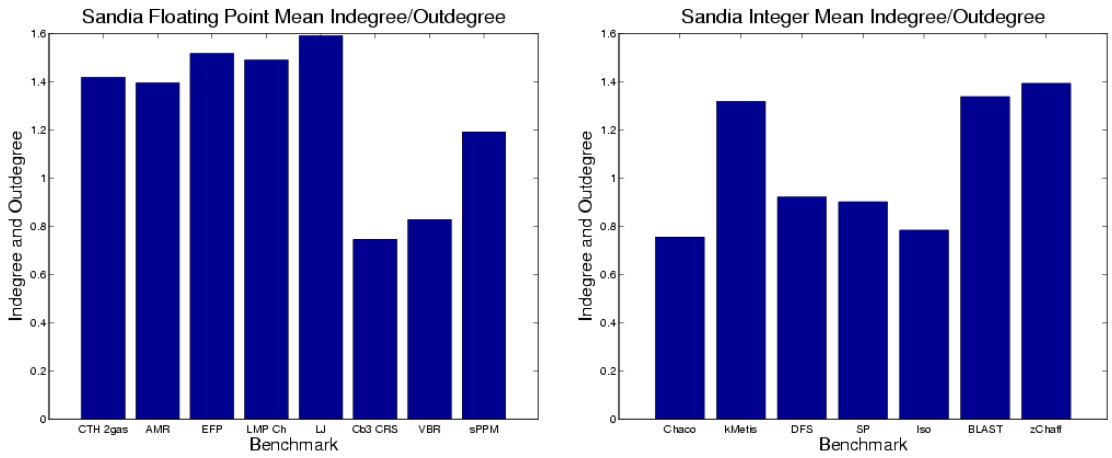


Figure 5.8. Dataflow Graph Data Reuse.

Figure 5.8 shows the mean indegree/outdegree for the benchmark suite, which is a measure of how often data is reused. In general, the particular results from a given vertex on the graph are reused less than 1.6 times before a new value is computed. The mean for the integer benchmark suite is 1.059, and for the floating point suite is 1.2724. This shows that relatively few vertices have high reuse: most computation results are consumed once. Full histograms of indegrees and outdegrees for each benchmark can be found in Appendix B.

Figure 5.9 shows the median indegrees for each vertex in the graph. This is one for every benchmark except CTH EFP and the two LAMMPS runs. Typically, half of all instructions consume one unique value. The median outdegree is zero in all the benchmarks, given that some values are stored to memory and never reused, and the instruction mix supports a large number of instructions that consume but never produce values (e.g., conditional branches, ALU operations using immediate values, and stores never produce data values).

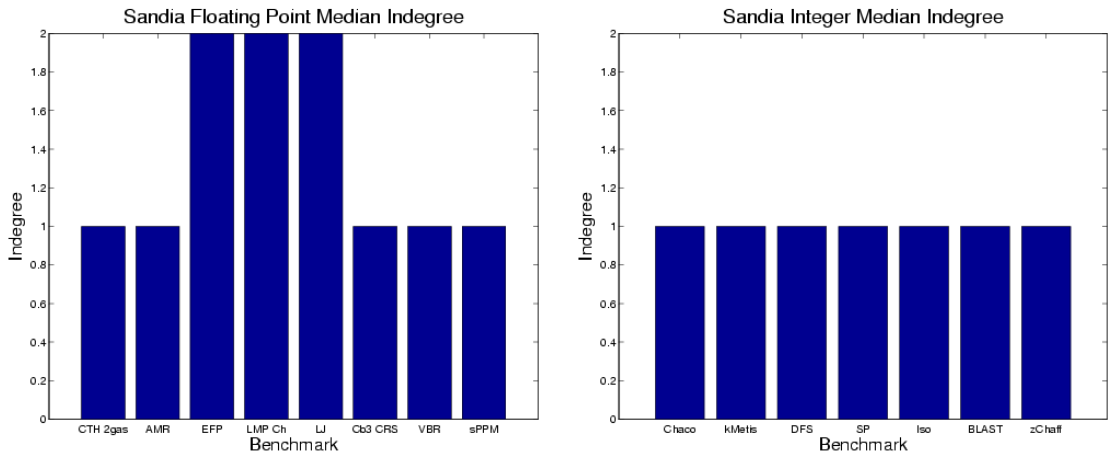


Figure 5.9. Dataflow Graph Indegree Medians

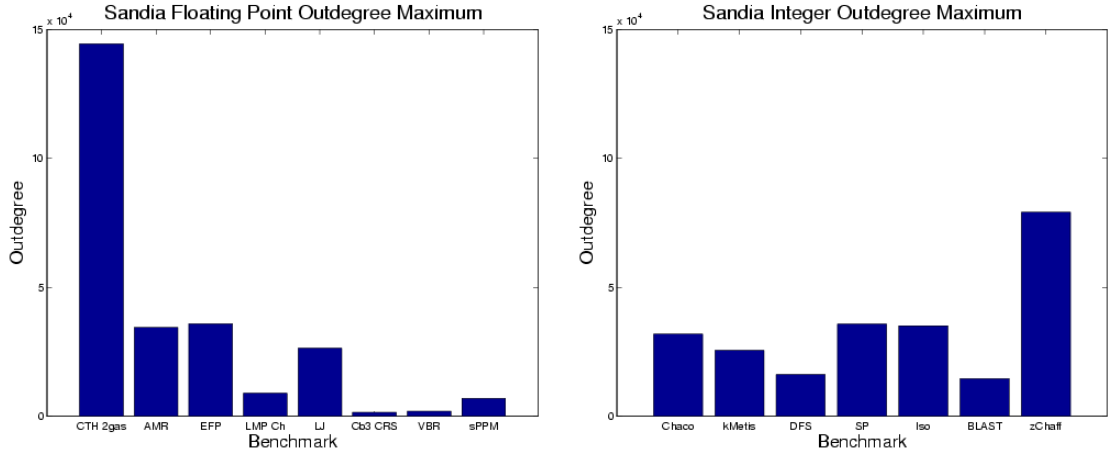


Figure 5.10. Dataflow Graph Outdegree Maximums

Figure 5.10 shows the maximum outdegree for each of the benchmarks. This reflects the maximum amount of reuse of any vertex's data values. The CTH 2Gas and zChaff benchmarks show significantly more reuse than every other. On average, the floating point suite shares a data value at most $32,559$ times, and the integer suite $33,998$. This confirms decades of computer architecture research demonstrating the

effectiveness of registers as a software controlled data cache, and experimental results showing that duplicating a very small number of frequently used read-only variables is highly effective at improving performance[95].

5.2.5 Data Longevity

Finally, the question of how quickly most results are reused is addressed. This is represented by the *use distance*, described previously in Section 5.1. The use distance, for every edge, is computed as the difference in the topological layer number between the edge's destination and its source. Because vertices typically have multiple edges, the use distance data presented in this section represents summary data. Appendix B contains histograms for each benchmark.

The use distance represents the number of topological layers between when a value is produced and when that value is consumed. In essence, it measures how far in the future a piece of data is needed, *for every time that data is used*. Specifically, if vertex v has edges (v, u) and (v, p), the data produced by vertex v is used twice (by u and p), and each edge has a different use distance. If (v, u) is used in the next topological layer (e.g., with a use distance of 1), and (v, p) is used 500 topological layers in the future, the mean use distance for vertex v's data value is 250.5. Critically, the topological layering removes artificial compiler interleaving of computation. That is, the dataflow represents a chain of computations required to get the answer. The compiler may be pursuing several such chains simultaneously, and the use distance as reported here removes any artificial interleaving (e.g., due to the register allocator, specific pipeline performance parameters, etc.).

The median use distance is always 1.0. This indicates that most data values computed can be used immediately (e.g., by an instruction in the next topological layer), while again, a minority of values will be used far in the future.

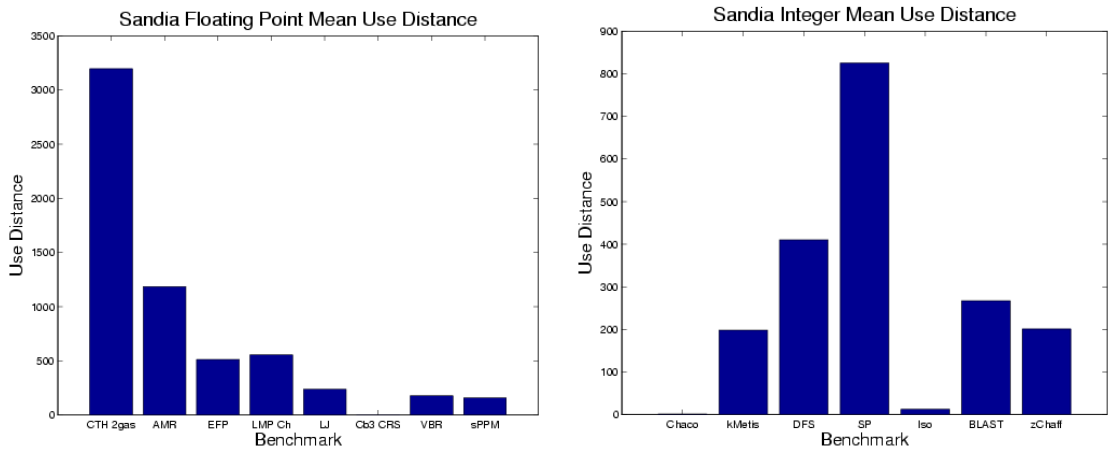


Figure 5.11. Mean dataflow use distance for all data item uses and reuses.

Figure 5.11 shows the mean use distance for each of the benchmarks, which is significantly larger than the median (of 1.0) because a minority of data values are used far in the future. It fundamentally represents the typical length of time a particular data item is needed (although it will likely be consumed immediately, it may be also consumed far in the future). Once again, CTH 2gas, LAMMPS LJ, DFS, Shortest Path, and Isomorphism stand out as reusing some data relatively far in the future. The mean for the suite is 273 time units for the integer suite, and 752 for the floating point suite.

Figure 5.12 shows the maximum use distances (inside the 1 million instruction stream). This represents the largest time interval over which a data item is reused. The mean for the floating point suite is 42,404 time units, and 37,997 time units for the integer suite. This is nearly the entire average critical path for the integer suite (41,149) but only about 60% the critical path for the floating point suite (70,118).

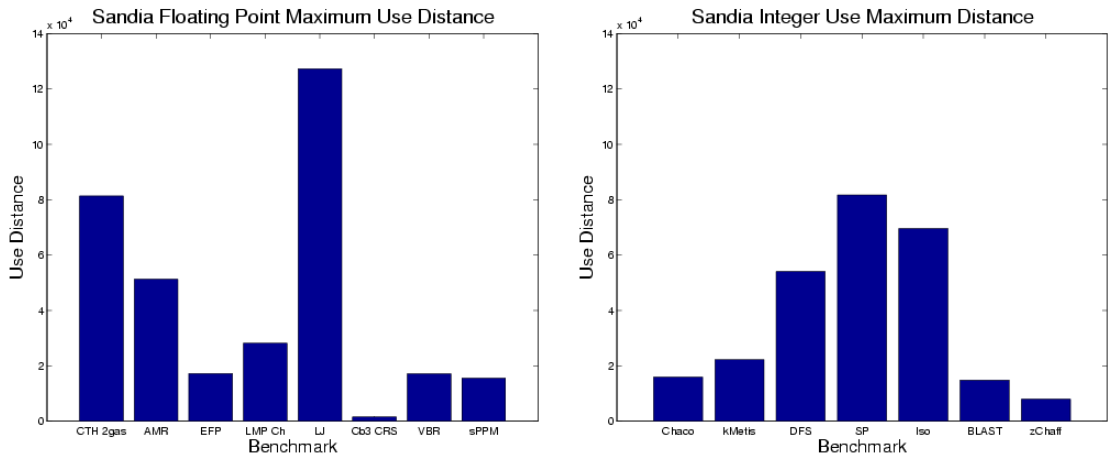


Figure 5.12. Dataflow Maximum Use Distance

5.3 Conclusions

The results show that most codes consume (and produce) a relatively small number of unique values, and that once a value is produced it is quickly consumed. Furthermore, an average of one to two orders of magnitude more concurrency should be available from the instruction streams as compiled today, providing ample opportunity for faster program execution and more latency toleration in software. In terms of small threads: most values are produced and consumed relatively quickly (and locally), indicating that small mobile threads can exist with very tiny state. In fact, despite the promulgation of RISC architectures with very large register files, and flexible multi-argument instructions, on average only 1.6 unique values is consumed per instruction. Some values, however, are heavily reused. These values tend to be infrequently written (because of their long use distance and the large number of inputs to the graph).

CHAPTER 6

THREADS

This chapter discusses the properties of threads, with a particular focus on thread synchronization. A *synchronization* occurs when one thread produces a data value that another thread will consume. Within a thread, when one instruction produces a value used by another instruction, no synchronization is required because instructions are executed in order. However, between two threads, when one thread produces a value that is consumed by another, no such execution order guarantee can be made. Consequently, the threads must synchronize so that data values are consumed only after they are produced.

Using the dataflow graphs from Chapter 5, which represent single-instruction threads, this chapter builds multi-instruction threads by using *minimum cut graph partitioning*, which cuts a graph into partitions which each contain an equal number of vertices while minimizing the *cut* between partitions. The *cut*, in this case, is the number of edges passing between two partitions. Since edges in the dataflow graph are always between producers of a data item and consumers of that item, they fundamentally represent required synchronization points between threads. In the case of the single-instruction threads discussed in the previous chapter, these synchronizations occurred via the topological scheduling, since an instruction in a given topological layer could not be executed before the instructions in all the previous topological layers. However, as threads are constructed from larger numbers

of instructions, these synchronizations occur explicitly every time an edge passes between two partitions. The primary focus of this chapter is on the number of synchronizations and the amount of unique bandwidth passed between threads, without regard to data placement, which will be examined in detail in Chapter 7. As a secondary focus, an upper bound of the amount of thread state required to perform a thread migration is determined.

In addition to the multithreaded architectures and runtime environments discussed in Chapter 2, there is an extensive literature on multithreading in the architecture, compiler, and operating systems communities. Program slicing[103, 37, 55] is used to decompose a program into independent parts determining which subsections of the program can be executed independently (because they have no interrelated values). The problem of identifying potential concurrent instructions has been extensively examined from both the control flow [68] and dataflow[80, 53, 31, 47] standpoints, with significant attention being paid to multithreading loops[45]. Beyond identifying parallelism, scheduling is of particular importance[18]. The Threaded Abstract Machine (TAM) model[34], specifies mechanisms for scheduling, synchronization, and thread management that are primarily under compiler control. This work focuses on the potential for creating threads via dataflow dependencies.

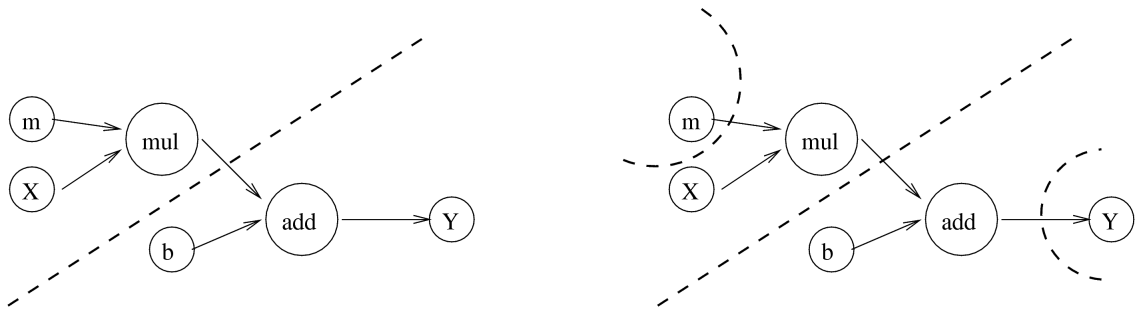
The remainder of the chapter is organized as follows: Section 6.1 discusses the graph partitioning methodology, and how it constructs threads with minimal synchronization between them, Section 6.2 discusses the experiment, Section 6.3 presents the resulting synchronization requirements from the threads constructed, and Section 6.4 ends with the conclusions.

6.1 Minimum Cut Graph Partitioning

The *minimum cut graph partitioning* problem is used to construct the threads required by this dissertation, as it provides a rational lower bound for the synchronization requirements of threads. Although this partitioning can be applied to any graph or hypergraph[1], in this case, the problem is: given the dataflow graph $G = (V, E)$, divide the graph into n (nearly) equally sized partitions, while minimizing the *total cut*, or the total weight of the edges that cross partitions. In the case of the dataflow graph, each edge has an edge weight of 1, so the minimization is of the number of edges crossing partitions. Edges represent the producer-consumer relationship between threads. That is, each directed edge (u, v) represents a data item produced by the instruction u and consumed by instruction v. By minimizing the total cut, the number of synchronizations between threads is minimized (since a synchronization is required between u and v for every edge because the value of that edge is not produced until u is executed). The minimum cut partitioning problem is NP-complete, but, because of its importance in areas such as VLSI circuit design and adaptive mesh refinement, several good heuristics exist, including METIS[59] and Chaco[50] (which are part of the integer benchmark suite under study), and the Fiduccia-Mattheyses Heuristic[27] often used in VLSI netlist partitioning. In this case, the METIS heuristic was used.

Figure 6.1 shows a 2-way and 4-way ($n = 2$ and $n = 4$) partitioning of the dataflow graph example discussed in Chapter 5. The cut is represented by the dashed line passing through the graph's edges. In Figure 6.1(a), there are two threads, each consisting of three instructions. The first thread contains a load of m and X, as well as the multiply, and the second thread contains a load of b,

[1] A *hypergraph* is a graph that contains *hyperedges*, which can connect more than one vertex. For example, given the vertices a, b, and c, a single hyperedge (a, b, c) can connect all three.



(a) a 2−way partitioning

(b))a 4−way partitioning

Figure 6.1. Example (a) 2-way and (b) 4-way partitioning of the dataflow graph.

the add of b and the results of the multiply instruction (which requires a single synchronization), and a store to Y. This partitioning is well balanced (there are an equal number of instructions in each of the two threads), and contains the minimum possible total cut of 1. Figure 6.1(b), shows the partitioning of the same graph into 4 parts, where the load of m and the store to Y are separated into additional threads. This partitioning retains the minimal cut (3), but is not unique (the load of X or the load of b could have been chosen instead for the same minimal cut), and is less well balanced since there are two threads of 1 instruction and two threads of 2 instructions. This is to be expected given the very small nature of the example graph.

6.2 Experimentation

In this experiment, each of the dataflow graphs is modified slightly from the graphs given in Chapter 5. In the original graphs, there explicitly exist *sources* and *syncs*, that is, memory locations from which data is loaded, and to which data is stored at the beginning and end of the instruction stream. The *sources* are of

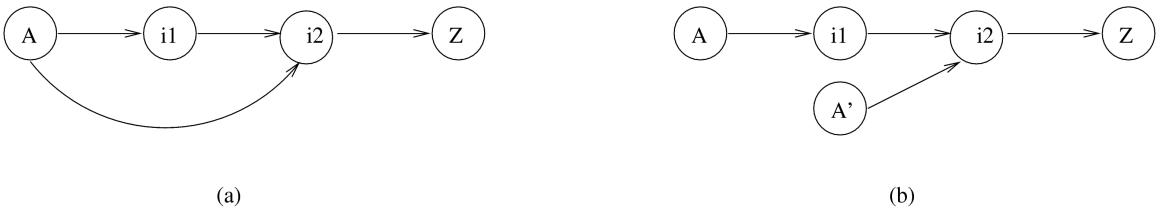


Figure 6.2. (a) the original dataflow graph and (b) the modified version (b) that allows the cloning of read-only data from memory.

particular importance, because often times data values from memory are read-only, and, in that case require no synchronization. Naturally, these values are only read-only for the length of the instruction stream under study (that is, they may be updated after the $1,000,000$ instructions under study have executed).

Figure 6.2(a) shows an example with a single memory source A and memory sync Z. The unmodified value of A is used both by instructions $i1$ and $i2$, and in the original dataflow graph two edges were required. Figure 6.2(b) modifies the dataflow graph so that a new vertex A' (which is a copy of A) is created to represent the fact that no synchronization is required given that A has not been modified. This was undesirable in the original dataflow graph because the first version (Figure 6.2a) better shows the *reuse* of unique data items, whereas the modified graph (Figure 6.2b) reflects the fact that, because A is a read-only variable (and thus, never modified) it never requires a synchronization. In both cases, if the data crosses the partition between two threads, it is *shared* in that a producer created the value for use by a consumer. However, the data's *reuse* characteristics are different. In the former case, A is reused twice, whereas in the latter it is copied and the original A and its copy are each used only once. This can reduce the required number of synchronizations, because only values that may be written to require a synchronization.

The modified 1 million instruction dataflow graphs are then partitioned into 100,000, 10,000, 1,000, 100, and 10-way partitions, which produce threads varying in length from 10 to 100,000 instructions (threads of 1 instruction were discussed in Chapter 5). This permits the study of synchronization scaling as the thread size varies, which is of critical importance because synchronization on most conventional architectures is a very expensive proposition. On more unconventional architectures, such as the MTA and its descendents, synchronization is made cheap in time, but expensive in space (with 2 bits of overhead for full/empty bits for each word in memory). The question addressed here is more general: what is the trade-off between thread-length and synchronization events.

Finally, each thread created via the graph partitioning is examined to determine the number of unique registers used (either via a read or a write to that register) by that thread. This, along with any non-programmable visible state (e.g., a program counter and thread status word) is the upper-bound for the thread state required to move since it represents what the compiler actually required for register allocation. It is possible that the compiler could do better by freeing a register allocation after it is no longer needed, or that the compiler could decrease the required number of allocations and increase the thread length by spilling register values to memory, however that is left for future work.

6.3 Results

This section describes the synchronization requirements of the threads created by partitioning the dataflow graph. This represents the lower-bound as synchronizations occur when an edge in the dataflow graph is *cut* (e.g., passes between two partitions), and the minimum-cut partitioning mechanism optimizes the cut. The cut results are presented in terms of the number of data items *produced* by

one thread for another, or, symmetrically the number of data items *consumed* by a thread. For any edge (u, v) which crosses between two partitions (or threads), the partition containing u produced the data item represented by the thread, and the partition containing v consumed that data item.

Appendix C shows the individual results for each benchmark in terms of a histogram of synchronizations, the percentages of threads requiring that no other threads produce inputs for them (e.g., read entirely from unmodified data values in memory), and the number of threads that produce no outputs for other threads in the dataflow graph (e.g., written to memory but not consumed in the current graph). A given thread's *inputs* are dataflow edges that cross into the thread from another thread (and are consumed). Similarly, a given thread's *outputs* are data flow edges that the thread produces and are consumed by another thread. The remainder of this section presents averages over each of the two benchmark suites as summary results.

6.3.1 Mean Synchronization

The mean synchronization is given by the mean cut between partitions. As with previous chapters, the mean is the mean for both producers and consumers because any edge, (u, v), in the dataflow graph must be between a producer u and a consumer v.

Figure 6.3 shows the mean number of (machine length) words requiring synchronization on both a per-thread and a per-instruction basis. The floating point suite requires, on average, 12.7× the number of synchronizations per instruction for a 10 instruction thread than for a $100,000$ instruction thread, and the integer suite 17 times the number of synchronizations per instruction given the same thread lengths. Threads of shorter length require more thread synchronizations on a per instruction

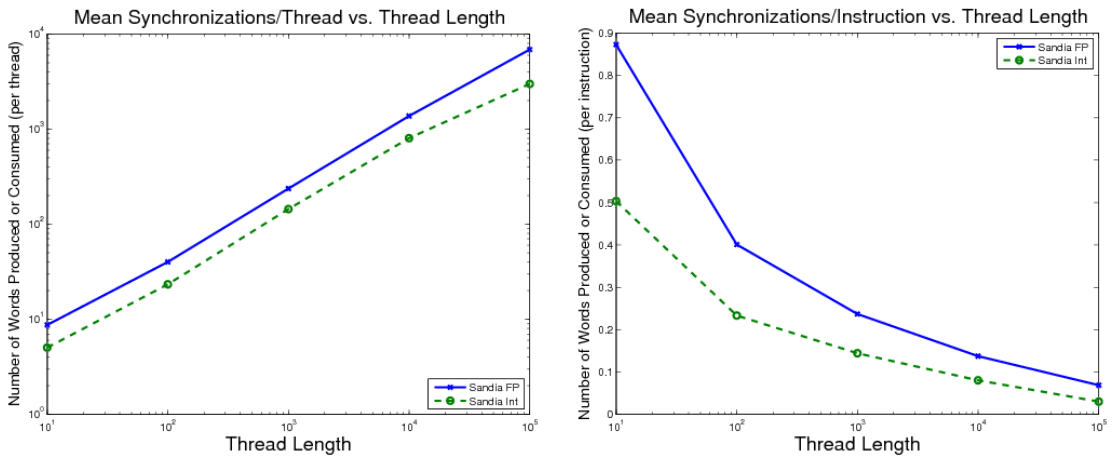


Figure 6.3. Mean Thread Synchronization Requirements on a Per-Thread and Per-Instruction basis.

basis than do threads of longer length. This is because data values that are produced in longer threads are more likely to be consumed within the same thread. In terms of 10 instruction threads, the 0.8725 and 0.5031 synchronizations per instruction required by the floating point and integer suites respectively are significantly less than the 4 synchronization maximum that would be required if every data item for an instruction using the maximum number of registers required a synchronization. Furthermore, since the dataflow graph is represented as a directed acyclic graph (as opposed to a hypergraph, where reuse between multiple vertices would be represented by a single hyperedge), reused data values that are the same each have their own edge. That is, if a given thread, A, produces the value X, which is required by two threads, B, and C, then X actually produces two edges: (A, B) and (A, C). This allows for simplicity of representation in the dataflow graph, as well as simplicity of counting reused data items. It also, critically, preserves ordering for scheduling.

There is a significant drop in the required sharing between 10 and 100 instructions. The floating point suite requires 2.2× the number of synchronizations per instruction as the thread length is decreased, and the integer suite 2.16×. Depending upon the implementation of the synchronization mechanism, as well as the amount of concurrency available, this may be a better trade-off point as it is the single largest proportional drop (by over 25%). However, this is entirely dictated by the implementation. As expected, the results demonstrate that contemporary superscalar processors with relatively slow synchronization mechanisms would pay a tremendous penalty for significantly decreasing the thread length. In the case of the PowerPC's reservation mechanism (as discussed in Section 4.2.5), the architecture simply does not allow enough outstanding reservations to support the nearly one per instruction required. Furthermore, the loads and stores that perform reservations require a test for success to determine if they actually completed (so that the programmer may poll). Even if this test is always successful, the additional branch required to perform the test would nearly double the number of instructions in the thread. And this only accounts for synchronizations through memory (which are about 40% of instructions in these codes). A new instruction would have to be created to perform synchronization through registers. Thus, in a 10 instruction thread, which generally requires 9 synchronizations for the floating point suite, 4 of those instructions access memory, and could use the PowerPC's reservation mechanism. If that mechanism supported an unlimited number of reservations, the programmer would still have to perform 4 additional tests to determine if the reservation succeeded. Of the 6 remaining instructions, 5 would require synchronization through at least one register. If that synchronization required the same type of test as the synchronizations through memory, 5 additional branches would have to be introduced into the thread. This constitutes an increase in the number of instructions

of over 87% for the floating point suite. The same analysis for the integer suite requires more than a 50% increase in the number of instructions. Consequently, the synchronization mechanism has to be extremely light weight merely to satisfy the producer/consumer relationships between threads, particularly when the thread lengths are short.

6.3.2 Median Synchronization Values

As with the results in the previous chapter, the medians prove better than the means in differentiating the producers from consumers. In this case, the medians are the mean of the medians for a given benchmark suite (the individual medians can be found in Appendix C). That is, the medians are computed individually for each benchmark, and the median for the suite is given as the mean of the medians.

Figure 6.4 shows each benchmark suite's median for consumer and producer synchronizations on both a per-thread and per-instruction basis. As with the means, the biggest relative difference in consumer synchronizations comes between threads of 10 and 100 instructions. Additionally, the integer benchmark suite requires about half the consumer synchronizations of the floating point suite. In general, there are far fewer unique values produced than consumed, indicating a very high degree of reuse, as well as a large number of read-only values coming from memory. In fact, there is over a 6× difference for the floating point suite, and over a 3.5× difference for the integer suite. Since the dataflow graph tracks instructions producing new information and all the places where that information is consumed, this is indicative of a relatively small amount of new information being produced. Read-only information originating in memory, due to the construction of the graph, is cloned wherever its value is read.

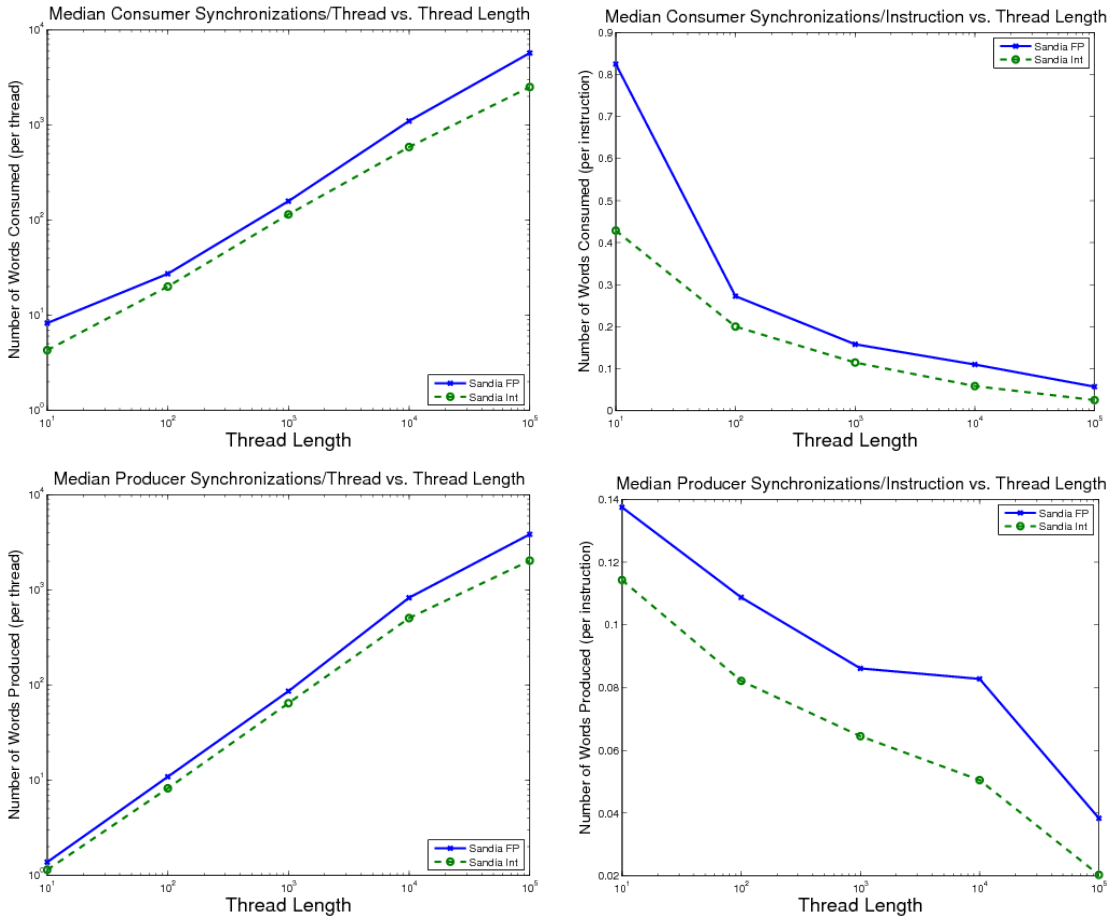


Figure 6.4. Median Consumer and Producer Thread Synchronization on a Per-Thread and Per-Instruction Basis.

6.3.3 Internal Computation

Finally, it is important to know whether or not a large percentage of threads consume their inputs only from read-only memory, or produce their outputs only to memory which is not read again in the instruction stream under study.

Figure 6.5 shows the percentage of threads that either take all their inputs from read-only memory, or write their outputs to memory which is not read again in the dataflow graph. The results show that, until thread lengths of $100,000$ are

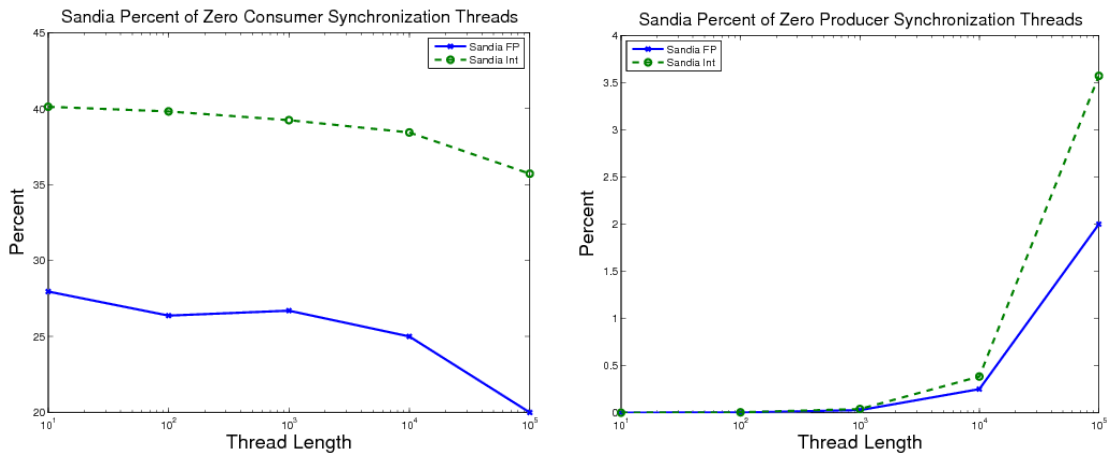


Figure 6.5. Threads that produce or consume their data from memory, requiring no synchronization.

reached, over 25% of the threads in the floating point suite and nearly 40% of the threads in the integer suite take their inputs solely from read-only memory and require no synchronization. This is particularly beneficial for short threads as they are potentially capable of bringing significant parallelism to the application at very little cost. Specifically, these threads can be started at the beginning of the instruction stream because all their data is available. This is largely a result of the fact that there exists a lot of unutilized concurrency in existing instruction streams. Again, for the most part, this input ratio is nearly constant across threads. It is almost a uniform 40% in the integer suite, while the floating point suite shows a bit more variation.

Outputs, on the other hand, are much more likely to be consumed than written to memory and never accessed again. The only threads that tend to consume most of their own data are the $100,000$ instruction threads, and then no more than 3.5% of them (on average). As expected, the shorter instruction streams tend to pass their results onto additional threads for analysis.

TABLE 6.1

THREAD REGISTER STATE REQUIREMENTS

Suite	**Result**	**Thread Length**				
		10	100	1,000	10,000	100,000
Sandia FP	Number of Registers	10.33	23.96	31.17	38.14	41.54
	Required Bytes	96	200	264	320	344
	Cache Line Ratio	0.75	1.56	2.06	2.50	2.69
Sandia Int	Number of Registers	7.75	15.58	17.91	19.45	22.01
	Required Bytes	72	136	152	168	192
	Cache Line Ratio	0.56	1.06	1.19	1.31	1.50

6.3.4 Thread State

In this section, each individual thread created via partitioning is examined to determine the number of unique registers that thread consumes. This represents the results of the PowerPC compiler's register allocation effort, where the compiler attempts to optimize memory references into 32 32-bit integer registers and 32 64-bit floating point registers. Since the PowerPC is not compiling with any attempt to optimize for a small number of registers, this is a realistic upper bound on the required thread state to perform a migration.

Table 6.1 shows the mean number of registers required by each thread length (10-100,000) for the Sandia Floating Point and Integer Suites, as well as two key derivative values:

- **Required Bytes:** which represents the number of bytes required to transfer the contents of the programmer visible registers (rounded up to the next register), plus one thread status register, assuming each register was 64-bits.
- **Cache Line Ratio:** the bandwidth requirement as a ratio to a 128-byte (L2-sized) cache line, which is computed as $\frac{RequiredBytes}{128}$ (where the number of required bytes is defined above).

The results show that small threads require a relatively large number of unique registers: a 10 instruction thread requires a mean of 10.33 registers for the floating point suite and 7.75 registers for the integer suite. There are two reasons why this number seems relatively high given the low number of instructions: first, the very short threads do significantly more sharing of data through registers than through memory locations; and second, the PowerPC is (for the most part) a 3-register register architecture. Given the instruction mix (with about 50% of instructions performing computation, which would typically use 3 registers), the fact that approximately two unique registers per instruction are required for very short threads is not surprising. It is likely that a different ISA or different register allocation algorithm would decrease this demand.

As the thread length increases, obviously more register state is used. When compared to a caching architecture, threads of 10-100 instructions consume approximately a cache line of state (or less) in the integer suite, and threads of 10 instructions consume less than a cache line of state. If the number of thread migrations for the 10 instruction threads can be made less than the number of cache transactions, less aggregate bandwidth will be required of a traveling thread[2]. This bandwidth savings can be achieved while simultaneously *halving* the number of high-latency network transactions (as a traveling thread migration is a one-way network transaction rather than a request-response transaction). Chapter 7 examines the aggregate number of migrations required of traveling threads given different data partitionings.

[2]This estimate excludes the bandwidth required to initiate the memory transaction (e.g., the request), and counts only the data bandwidth consumed in the response to that request.

6.4 Conclusions

As asserted in Chapter 4, short threads require a very light weight synchronization mechanism to perform effectively. At thread lengths of 10 instructions, nearly 9 of them will require some sort of input or output synchronization. Thread lengths of 100 instructions seem to offer an optimization point requiring fewer synchronizations, but the effectiveness of this point depends on the cost of the synchronization mechanism. In smaller thread lengths, inputs tend to come reasonably frequently only from read-only memory: that is memory values presented to the dataflow graph at the beginning of time are often unmodified throughout the stream and exhibit significant reuse. This intuitively makes sense in that loads outnumber stores in the instruction mixes. Furthermore, as expected, shorter threads tend to produce relatively more "intermediate values" that pass through to additional threads as the instruction stream is partitioned. A large amount of thread-level concurrency is available in the instruction stream, just as a large amount of instruction-level concurrency was available in the dataflow graphs. This is confirmed by the relatively large proportion of threads (particularly smaller run-length threads) that require no synchronized input to start. Finally, the thread state, regardless of thread length, varies between 0.56 and 2.69 128-byte cache lines (depending on thread length), indicating that small threads can have their state encapsulated in a relatively small number of bytes (which, in turn, will be transmitted from one node to another during a thread migration).

CHAPTER 7

DATA PARTITIONING

This chapter addresses the issue of data partitioning. In a parallel environment (regardless of whether or not the memory address space is shared), *data partitioning* assigns data items to a memory node in the computer. Choosing the "right" node has tremendous potential for performance improvement. By increasing the spatial locality of all the data items (globally, or optimized against each other), the number of remote data accesses (either explicit or implicit) is reduced. For example, in the simple calculation $Y = mX + b$, if Y, m, X, and b all reside on the same node in a parallel computer, no remote (or high latency) data references are required. This is true for *both* conventional caching architectures and traveling threads. If, on the other hand, any of the data items do not reside on the same node (in the worst case, each data item resides on a different node), those data items must be fetched and cached, in the case of a conventional caching architecture, or exchanged explicitly between nodes in the case of an MPP, or cause thread migrations in the case of the traveling thread architecture. By placing the data such that the spatial locality is improved (globally, across the parallel machine), the number of these high latency transactions (regardless of their type or weighting) can be reduced. For traveling threads, the purpose of improving the data placement is to increase the data's spatial locality and decrease the number of thread migrations.

The programmer tends to allocate memory contiguously in the virtual memory space. The program's heap starts with the variables statically allocated by the programmer (generally in the order they were declared), then proceeds to contain dynamically allocated data structures (generally, in the order that the allocation was requested). On many full-scale operating systems, the physical mapping of these addresses is a complex matter: it depends on the other processes running in the system, the amount of free physical memory, and whether or not the operating system uses a *page coloring algorithm* to improve cache performance by choosing a physical placement in memory meant to avoid cache conflicts with other pages in the program. On a supercomputer operating system, such as the lightweight kernel (see Section 2.4.2), the memory allocation is a simpler, contiguous allocation with a corresponding mapping between virtual and physical pages. This chapter examines the degree to which improving the programmer's contiguous (naive) allocation could improve program performance.

Improvements in data placement are essential tools in reducing program execution time because remote data accesses are significantly farther away than local data accesses. In this work, data partitioning is done by constructing the *data transition graph*, in which vertices represent locations in memory, and edges represent the temporal transition between those locations (e.g., each edge weight between u and v represents the number of times the program accesses memory location u followed by memory location v). Each transition (e.g., referencing memory location u followed by memory location v, which creates the edge (u, v)), represents a traveling thread migration if u and v are placed in different data partitions. By applying minimum-cut partitioning to this graph, as described in Section 6.1, the number of traveling thread transitions between different nodes in the machine is minimized. When an edge in the data transition graph passes between two parti-

tions, the edge weight represents the number of times a traveling thread must pass between those two partitions in search of its data. By minimizing the total weight of the edges in the data transition graph that pass between two data partitions, the number of high-latency traveling thread migrations has been minimized. This chapter compares that minimized transition count to what would be required by the programmer's given partition (representing no change in data partitioning), and a random partitioning (representing a very bad partitioning).

Prior work for data partitioning has focused on a very different programming model: either the request-response model of modern cache coherent shared memory machines, or the programmer-specified data exchange model popular in message passing architectures such as MPI. In any case, the goal is the same: to cluster the set of data values needed to perform a computation on a given node together to increase their spatial locality. In the field of parallelizing compilers, attempts to automatically determine the data partitioning are numerous [44, 92, 39, 91, 93]. Programming languages such as High Performance Fortran[54], or Paradigm[14] include explicit mechanisms for programmer specified data placement. Additionally, there are numerous dynamic techniques for partitioning the data at runtime, such as adaptive mesh refinement (AMR), that take advantage of the structure of a parallel computation to perform better load balancing[49, 43, 88, 16, 15]. This chapter examines the unique partitioning requirements of a traveling thread in terms of the potential for decreasing the number of thread migrations that need to be performed to realize the computation that is specified by a given thread.

The remainder of this chapter is organized as follows: Section 7.1 describes both the serial and parallel computation versions of the data transition graph; Section 7.2 explains the partitioning of the data transition graph; Section 7.3 presents the results of the data transition graph partitioning; Section 7.4 uses the partitioning

results to analyze traveling threads of various lengths; Section 7.5 compares the traveling thread results to a conventional processor; and, finally, the conclusions are given in Section 7.6.

7.1 Data Transition Graph

In the data transition graph, vertices represent locations in memory, and weighted edges represent the number of times a transition from one memory location to another is made. For example, if the program accesses memory location u followed (temporally) by memory location v, the edge (u, v) is created. If this happens 3 times, that edge is given weight 3. Because transitions from u to v and from v to u are equivalent in cost, the data transition graph is represented as an undirected graph, where edge weights represent transitions in either direction.

7.1.1 Serial Computation Graphs

The serial computation data transition graph is the simplest form, and represents the data transitions for a single, serial (e.g., in-order) computation. In other words, it represents the data transition graph for the serial instruction stream, executed in-order, before it was divided into threads.

Figure 7.1 shows the data transition graph for a simple loop, both as the programmer allocated the data (contiguously in memory), and using minimum cut partitioning. In the example, there are two equal sized partitions. The programmer tends to allocate data structures contiguously in memory: first a, then b, then c (in the example there are only two elements in each array). In the programmer's allocation, there is a cut of 3 for the graph, but that cut can be optimized to 1. This is used to optimize *latency*, as each transition represents a move from one node to another in the machine, which is a relatively long operation.

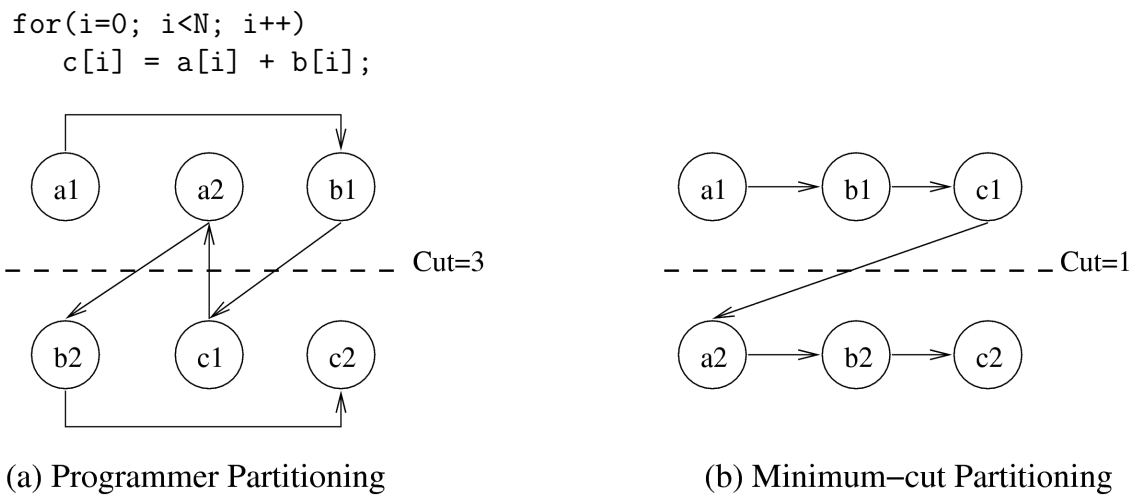
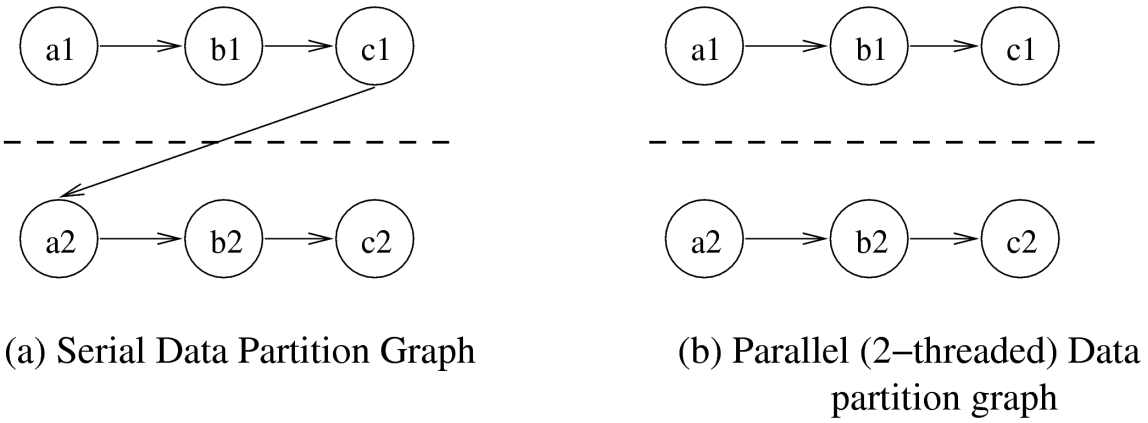


Figure 7.1. The data transition graph for the first two iterations of a simple loop. Part (a) shows the programmer's original (contiguous) memory allocation; and part (b) shows the minimum cut partitioning.

7.1.2 Parallel Computation Graphs

The previously described data transition graph only accounts for data partitioning of the serial instruction stream (before it is broken into multiple threads, as described in Chapter 6). When the same stream is broken into multiple threads, the graph is slightly different. In fact, even serial instruction streams can have multiple data transition graphs. As discussed in section 5.1.3, any thread can be executed in any valid topological ordering of that thread's dataflow graph, consequently different data orderings could be produced.

The difference between the serial and parallel versions of the data transition graph is shown in Figure 7.2. The original serial graph is shown in (a), and the same graph broken into two threads (one computing $c[1] = a[1] + b[1]$ and one computing $c[2] = a[2] + b[2]$) is shown in (b). The parallel version of the graph is computed by simply iterating through each thread's instruction stream separately, and producing one large graph. Typically, the parallel graph removes edges, and separates any



(a) Serial Data Partition Graph

(b) Parallel (2−threaded) Data partition graph

Figure 7.2. The Serial and Parallel Data Transition Graphs for two iterations of the loop from Figure 7.1.

interleaved computations chosen by the compiler. As mentioned previously, within a thread the ordering could be changed. For example, the addition operation is commutative, and $a[i] + b[i]$ could be computed as $b[i] + a[i]$, which would cause the vertices a_i and b_i to be swapped. For the purposes of this dissertation, all scheduling is done in the order of the layered topological graphs produced in Chapter 5.

7.2 Data Transition Graph Partitioning

This work focuses upon the serial computation data transition graph as a baseline for evaluating traveling threads. First, the serial graph is used to evaluate the potential improvement over programmer provided (contiguous) and random partitioning through minimum-cut partitioning. Then, the minimum-cut data partitioning is used to evaluate traveling thread migration as the number of data partitions and the thread length are varied. The parallel computation data transition graph for each set of threads could have been used to improve performance (in terms of the number of migrations), but would not allow for a comparison of thread migration over a single data placement as the length of the thread was varied.

The serial computation data transition graph is constructed by examining 100 million instructions from the same instruction traces that have been used throughout the rest of this work. This graph uses two orders of magnitude more instructions than the dataflow graph so that data partitioning can be examined in the context of the larger computation. The size of the data transition graph is still restricted by the practicality of partitioning large graphs. The threads produced in Chapter 6 are therefore a subset of the computation represented by the data transition graph. The serial data transition graph is then partitioned into 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 parts using three methods:

1. **Minimum Cut Graph Partitioning:** which minimizes the weighted edge cut between partitions (or the number of times a traveling thread travels between two partitions);
2. **Programmer Order:** where every data item that the program consumes from the data transition graph is placed into 8 KB pages (to eliminate large, unaccessed regions of memory) and allocated to each partition contiguously.
3. **Random:** where the partition for each data word in memory that the programmer uses in the data transition graph is distributed among the number of partitions using a random number generator.

These partitionings represent an optimized, programmer specified, and unoptimized partitioning respectively. The critical result from this chapter is given by the *cut* of the graph as it is partitioned: or, the number of transitions that the serial thread under study must perform to successfully compute the result. In this case, there is no data caching, so that the thread moves every time a new piece of data is requested that is not on the current node upon which the thread is executing. Consequently, the true potential improvement contributed solely by improved data partitioning is determined. As in previous chapters, the *cut* is defined as the sum of the weight of the edges of the data transition graph that cross between partitions.

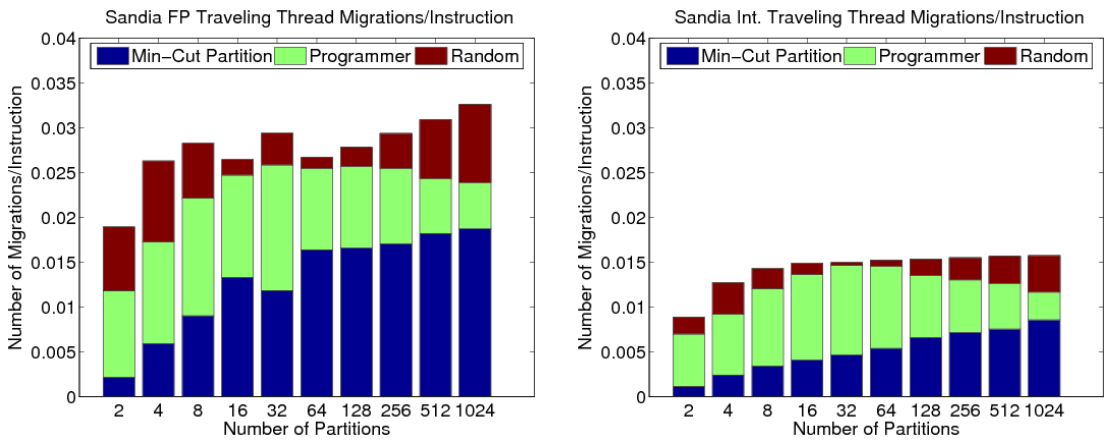


Figure 7.3. Sandia Integer and Floating Point Migrations per Instruction

7.3 Partitioning Results

This section describes the aggregate results of partitioning the data transition graph for the integer and floating point benchmark suites. Individual benchmark results are given in Appendix D.

Figure 7.3 shows the results of partitioning the floating point and integer suites as the number of migrations required per instruction for each of the three partitioning under study. It is immediately noticeable that the integer suite performs significantly fewer migrations (regardless of which partitioning is used) than the floating point suite. This indicates that, for a traveling thread, the integer benchmarks exhibit better spatial locality.

The results confirm that short threads have the potential to be very efficient, in that both the programmer and optimal partitioning migrate relatively rarely. For example, when partitioning the floating point suite into two partitions, the programmer's partitioning yields 0.0119 migrations per instruction and the optimized partitioning yields 0.0022 migrations per instruction. The programmer performs

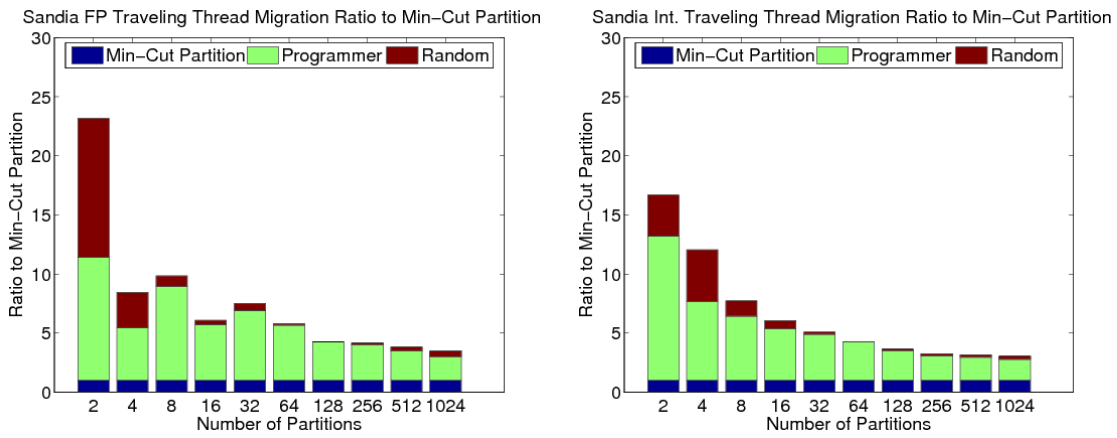


Figure 7.4. Sandia Integer and Floating Point Migration Ratios

over 5.4× worse than the minimum-cut partitioner. However, over the course of 100 instructions, the programmer will perform an average of less than one migration. The minimum cut partitioner will, on average, not perform a migration until over 450 instructions have been executed. Consequently, short threads, particularly when combined with improved partitioning, have the potential not to migrate often. Section 7.4 will examine thread migration after partitioning in significantly more detail, confirming this result. However, this result has powerful implications for reducing the number of high latency thread migrations the program must perform. Furthermore, given the results from Section 6.3.4, that show that shorter threads require significantly less state than do longer threads, this result shows the potential to provide a two-fold reduction in bandwidth requirements over previous traveling thread experiments: first, better partitioning yields fewer thread migrations (or network transactions); and second, shorter thread lengths require that less information be communicated over the interconnection network for each transaction.

Figure 7.4 shows the improvement that better partitioning yields as a ratio to the minimum-cut partitioning. The naive, contiguous partitioning (expressed by the

programmer) tends to perform relatively poorly (similar to a random partitioning). As the number of partitions increase (which divides the data between more nodes), the programmer's partitioning is relatively better, but there is still significant room for improvement. The programmer performs between 2 and 12.25 times worse than the minimum-cut partitioner, in terms of the number of high latency transactions.

7.4 Traveling Thread Results

Finally, in order to directly evaluate the effect of thread length on the number of migrations (e.g., events that consume interconnection network bandwidth and add remote access latency to the program), the threads obtained in Chapter 6 are combined with the optimized serial data transition graph partitioning described in Section 7.2. The full results for each benchmark are given in Appendix D.

Figure 7.5 shows the mean number of migrations per instruction for each of the threads created in Chapter 6 (from 10 to 100,000 instructions in length), and for the data partitioning obtained in this chapter (from 2 to 1024 partitions). The results show that shorter threads perform significantly fewer total migrations than do longer threads. That is, by splitting the same instruction stream up into more threads, the probability of a thread moving decreases. For example, in the Sandia Floating Point Suite, with 1024 partitions, 100,000 threads of 10 instructions each move only 41% the number of times that 10 threads of 100,000 instructions each move. Similarly, 10 instruction threads move only 58% as often as 100,000 instruction threads in the case of two partitions.

The results also demonstrate, as would be expected, that fewer partitions generally result in less movement. This confirms the results of Section 7.3, and also makes sense logically in that the smaller the number of partitions, the larger those partitions must be to support the same total amount of data.

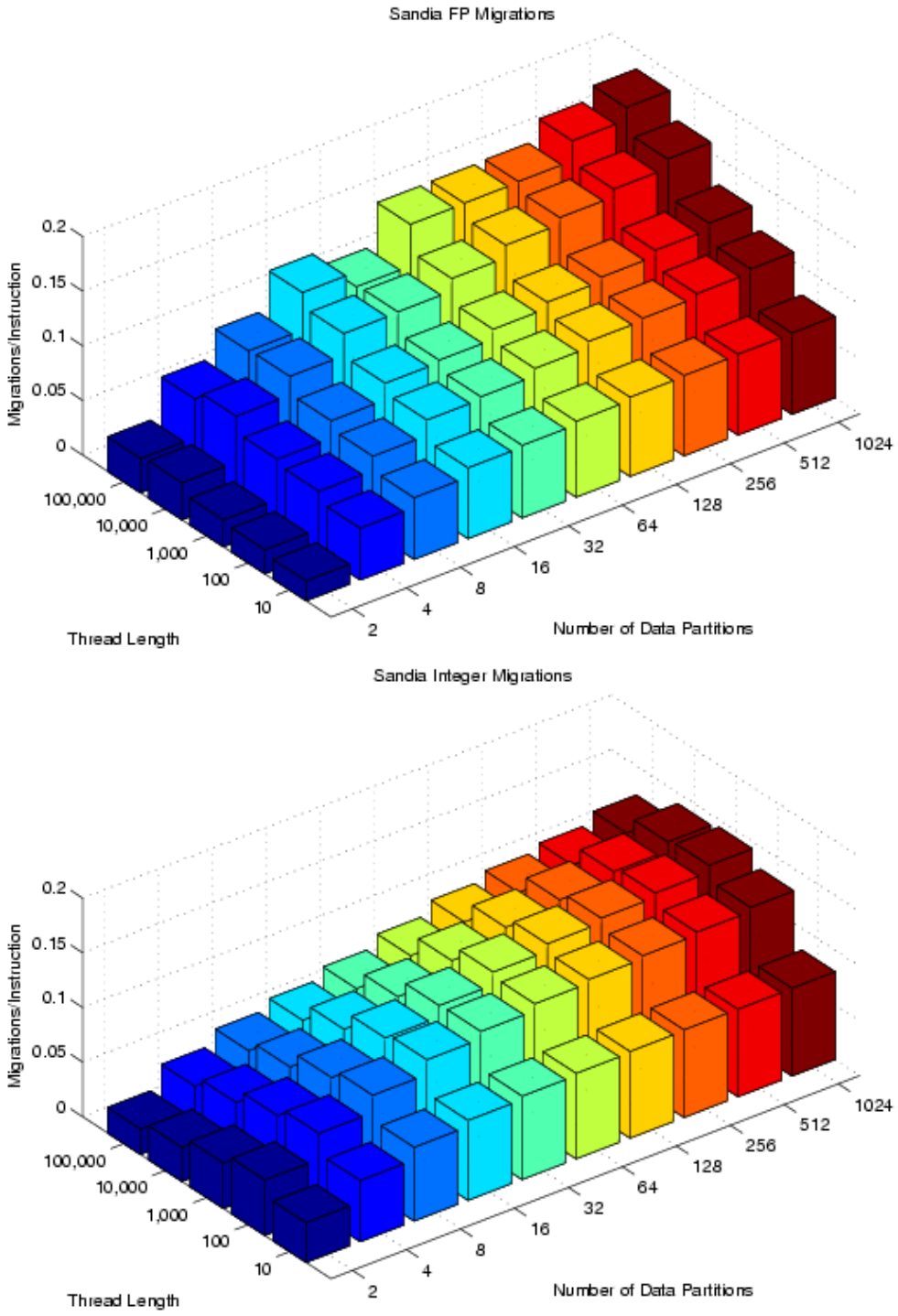


Figure 7.5. Sandia Floating Point and Integer Benchmark Suite Migration Summary.

7.5 Comparison to a Conventional Processor

In order to determine the success of the traveling thread model, it must be compared against a conventional processor. The most relevant comparison is execution time. The traveling thread program and the conventional program differ significantly only in the types of "high latency" events they perform, and the available concurrency. For the traveling thread, each thread migration causes activity external to the node. In the case of the conventional serial processor, this activity is caused by any cache miss generating a memory access.

The optimized traveling thread partitioning will be compared against the same computation running on a conventional processor with the following cache configuration:

- Level 1 Data Cache: 64KB, 8-way set associative, with a 64-byte block and a write-back consistency policy; and
- Level 2 Data Cache: 1MB, 4-way set associative, with a 128-byte block, and a write-back consistency policy.

This is typical of a modern microprocessor's memory hierarchy. Of interest, for the purposes of this comparison, are the number of transactions between the level 2 cache and the memory. These transactions are the high-latency event and occur under the following conditions:

- A read or write miss causes two transactions: a request and a response; and
- The write-back of a dirty cache line causes a single transaction (a write to memory).

Because caches are susceptible to compulsory misses when they start cold, the cache configuration described above was simulated for 1 billion instructions.

Figure 7.6 shows the L2 cache to memory transactions required of each of the benchmarks on a per-instruction basis. These are analogous to the migrations per

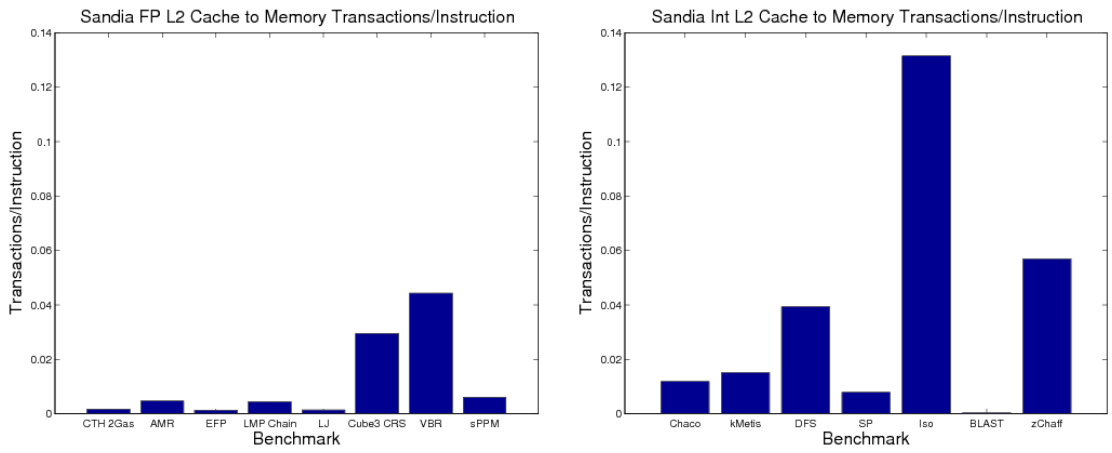


Figure 7.6. Sandia Integer and Floating Point Benchmark Suite L2 to Memory Transactions per Instruction

instruction given for the traveling threads in Section 7.4, as they are both the high-latency non-local events relevant to each execution model. Over the course of the entire program, the traveling thread model will perform more of these transactions, however the model also significantly increases the concurrency available to the application. Consequently, for an equivalent number of instructions, only the high latency transactions on the critical path of the program's dataflow graph contribute to increasing the latency. On the other hand, a conventional processor can only process as many memory transactions in parallel as can be placed on the memory bus simultaneously (in modern processors, 1-2 such transactions can be engaged in at a time). Furthermore, the modern processor can only issue the maximum number of those transactions that the bus will allow at all times if there exists sufficient instruction level parallelism within the stream it is examining to do so, that parallelism can be extracted by the processor's out-of-order execution unit. Assuming (very optimistically) that all the previously described conditions are met, the execution time for the conventional processor is bounded by:

$$\frac{N_{total} T_{memory/instruction}}{M} L_{memory} \tag{7.1}$$

Where N_{total} is the total number of instructions being executed, $T_{memory/instruction}$ is the number of memory transactions per instruction (as given in Figure 7.6), M is the number of memory transactions that the memory bus can sustain at any given time, and L_{memory} is the latency of a memory transaction. This boundary equation is relatively optimistic as it assumes that the conventional machine has enough available concurrency to sustain all M transactions on the memory bus continuously (e.g., it is the fastest possible execution). In the case of a highly concurrent set of traveling threads, the execution time is bounded by:

$$N_{criticalpath} T_{migration/instruction} L_{migration} \tag{7.2}$$

Where $N_{criticalpath}$ is the number of instructions on the critical path, $T_{migration/instruction}$ is the number migration transactions per instruction, and $L_{migration}$ is the latency of a thread migration.

$T_{memory/instruction}$ was measured in this section, $T_{migration/instruction}$ was measured in Section 7.4, and $N_{criticalpath}$ was determined in Chapter 5.

Because $T_{memory/instruction}$ accounts for memory accesses requiring a request and response (e.g., reads and writes) as two transactions, it can be assumed that $L_{memory} = L_{migration}$ (or the time to communicate on the interconnect for either machine is the same). If so, the only unmeasured value is M, which can be used to compute to determine when a conventional processor can equal the execution time of a traveling thread. Traveling threads execute in the same time as a conventional processor when:

TABLE 7.1

SANDIA FLOATING POINT BENCHMARK SUITE COMPARISON TO A CONVENTIONAL PROCESSOR

Thread Len	**Number of Partitions**									
	2	4	8	16	32	64	128	256	512	1024
10	9.2	3.5	3.0	2.6	2.5	2.4	2.3	2.3	2.3	2.2
100	7.6	3.0	2.4	2.0	1.9	1.8	1.7	1.7	1.6	1.5
1,000	6.8	2.7	2.2	1.8	1.8	1.6	1.5	1.5	1.4	1.3
10,000	4.9	2.2	1.8	1.5	1.5	1.3	1.2	1.2	1.1	1.0
100,000	5.3	2.5	1.8	1.3	1.4	1.1	1.1	1.1	0.97	0.92

$$\frac{N_{total}T_{memory/instruction}}{M}L_{memory} = N_{criticalpath}T_{migration/instruction}L_{migration} \tag{7.3}$$

Removing L_{memory} and $L_{migration}$ from the equation (because they are assumed to be equal) and solving for M such that the conventional processor has equivalent execution time to a traveling thread yields:

$$M = \frac{N_{total}T_{memory/instruction}}{N_{criticalpath}T_{migration/instruction}} \tag{7.4}$$

When the processor can support an M greater than that given in Equation 7.4 (assuming all the optimistic conditions given above), it could execute the same instruction stream faster than the traveling thread model.

Tables 7.1 and 7.2 show the computed values for M for the floating point and integer suites respectively. Most modern processors can issue one memory transaction at a time. Using point-to-point technology (such as hypertransport), some processors, such as the Opteron, can issue up to 3 simultaneous transactions, if they are independent. For the floating point suite, traveling threads of 10 instruc-

TABLE 7.2

SANDIA INTEGER BENCHMARK SUITE COMPARISON TO A CONVENTIONAL PROCESSOR

Thread Len	**Number of Partitions**									
	2	4	8	16	32	64	128	256	512	1024
10	25.1	16.3	13.5	12.5	11.9	11.7	11.5	11.4	11.4	11.4
100	18.0	12.2	10.1	8.8	8.2	7.8	7.6	7.4	7.2	7.0
1,000	22.0	13.8	11.1	9.2	8.3	7.5	7.1	6.9	6.7	6.4
10,000	29.4	17.1	13.6	11.1	9.8	8.5	7.7	7.3	7.0	6.5
100,000	39.4	20.9	16.0	13.8	12.0	10.5	9.2	8.5	8.0	7.4

tions, regardless of number of partitions, beat microprocessors that only support 2 simultaneous memory transactions. The integer results are significantly more dramatic, saying that, for any number of partitions, a conventional processor would have to support between 6.4 and 39.4 simultaneous memory transactions to equal the performance of the traveling thread execution model.

It should be noted that the computation of the value of M favors the conventional processor in that it assumes there exists sufficient instruction level parallelism to issue M transactions whenever possible. Furthermore, the comparison is against a uniprocessor conventional system and a multiprocessor traveling thread system. Were the conventional system to utilize more than one processor, Amdahl's Law would dictate that the performance would suffer. Minimally, coherency traffic would be generated between processors whenever any of the data is shared. This traffic would increase the demands on the memory system, requiring even more performance from the memory bus than is detailed in this result.

7.6 Conclusions

There is significant room for improvement in program data placement. In the data transition graphs studied, the programmer's contiguous 2-way partitioning of the data performs over an order of magnitude worse than the minimum-cut partitioning. In the worst case of improvement (1024 partitions), the programmer only performs twice as badly as the partitioner. Critically, a better understanding of which data items are inter-related could potentially unlock significant improvement in spatial-locality in an all-PIM traveling thread system.

When executing traveling threads over the data partitioning, smaller thread lengths significantly reduce the total number of network transactions that must be undertaken to provide sharing between threads.

Finally, conventional processors would have to support significantly more memory bus transactions than they currently do to equal the performance of the traveling thread execution model, regardless of the number of partitions. Again, shorter thread lengths yield better results in terms of exposing more concurrency and reducing the number of required external memory transactions.

CHAPTER 8

CONCLUSIONS

This chapter provides insight into the construction of a PIM-based traveling thread machine based on the execution model described in Chapter 4, and the experimental results described in Chapters 5-7. Section 8.1 describes the theoretical Traveling Thread Machine. Section 8.2 discusses some of the programming implications given the new execution model. Section 8.3 draws the conclusions. And, finally, section 8.4 describes the potential for future work in this area.

8.1 The Traveling Thread Machine

This section describes a new architecture, the Traveling Thread Machine, that implements the Traveling Thread Execution Model, and optimizes its architectural parameters in accordance with the results obtained in this dissertation to best improve performance. The choices for each of the execution model parameters in this section are from taken from the range of choices described in Chapter 4, and are supported by the experiments described in Chapters 5-7. The following inter-related pieces of the execution model must be addressed:

1. **Thread State Encapsulation:** represents the choice of how many unique registers traveling threads must have, how those registers can be used, and how they are represented by the architecture (e.g., in a register file, in memory, etc).
2. **Migration Mechanism:** is the choice of when and by what method a thread may migrate (e.g., whether or not all threads migrate, and if they do so implic-

itly because a remote memory reference has been made or explicitly because the thread has requested a migration).

3. **Synchronization Mechanisms:** addresses the question of how and through what architectural mechanism (e.g., registers, memory, or thread control flow) threads can share data that requires a producer/consumer synchronization as described in Chapters 4 and 6.

Because each of these choices is interrelated, to address each of the above areas, the following critical architectural questions need to be answered:

1. **What should the target thread length be?** The answer to this question is of three-fold importance:
   - First, the thread length affects the architecture in that inter-thread synchronization must be sufficiently light-weight to cope with the added cost of synchronizing as the thread length is reduced. Furthermore, the thread length affects the amount of state required for a thread to perform a migration (Chapter 6).
   - Second, the thread length affects performance, because short threads perform fewer thread migrations (e.g., they generate fewer high-latency network events), consume less bandwidth (Chapter 7), and expose more concurrency to the architecture (Chapter 6).
   - And Third, the target thread length serves as feedback to the compiler community, because it describes to them the type of concurrency they should attempt to find as instructions are being generated (to unlock the potential concurrency found in Chapter 5).
2. **How large a thread state will be supported?** The answer to this question impacts both the interconnection network designer, in that it describes how large of a network transaction a thread migration will be, and the compiler writer because they must perform register allocation for the thread that corresponds to the supported state size. Chapter 6 demonstrated that smaller threads require smaller thread state.
3. **What is the synchronization mechanism?** Because shorter thread lengths require significantly more synchronizations (on a per-instruction basis), the supported synchronization mechanism must be sufficiently light weight to support the required number of inter-thread communications without tremendously affecting execution time. This is of particular importance since most main-stream synchronization mechanisms are very heavy weight (Chapter 6).
4. **What is the migration mechanism?** Given the thread length chosen above, the migration mechanism must also be sufficiently light-weight so that execution time does not suffer.

Finally, the related question of **how can a machine with these properties best be programmed?** will be examined for the first time.

8.1.1 Thread Length

The first, and most important, architectural parameter to choose is the thread length, as it impacts all other architectural choices, and is most directly related to the program execution time. There are three competing architectural trade-offs relevant to the selection of thread length. The first, and perhaps most important, is the amount of available concurrency, as this most directly affects execution time (particularly in a latency tolerant architecture). The second, is the fact that as the thread length decreases, the number of inter-thread synchronizations increases. And, finally, as the thread length decreases, so does the thread context size required to perform a thread migration. It has been demonstrated that a shorter thread length exposes more program concurrency and reduces program execution time, at the cost of increasing the number of inter-thread synchronizations.

Chapter 5 showed that the shortest possible threads – one instruction – provide a potential median increase in concurrency of one to two orders of magnitude (or more) over the serial stream. That increase in concurrency must be balanced against the number of required synchronizations, and the thread's context size. From the perspective of latency toleration, increasing the number of threads that can be run at any given time has the potential to tolerate longer latencies. Furthermore, the improved program provides fewer migrations and requires less state for each migration (See Section 8.1.2 for an extended discussion of context size). Because this saves both latency and bandwidth, it is advantageous to create an extremely light-weight synchronization mechanism to support the additional synchronization requirements caused by breaking the program into smaller threads. Section 8.1.3 will discuss the synchronization mechanism in greater detail.

8.1.2 Thread Context Size

In terms of context size, because interconnection network bandwidth is at a premium, decreasing the context size is critical. A 128-byte context would allow for 15 64-bit general-purpose registers and 1 64-bit program status word (including a program counter and thread state private to the architecture, such as capabilities, exceptions, condition codes, etc.). However, in the case of very small threads (e.g., 10 instructions or less), only 56% to 75% of that state is actually required. Any wasted state consumes memory resources on the PIM and network bandwidth. On the other hand, longer threads require more register state. There are two implementation options:

1. Allow threads to have multiple thread state sizes (chosen by the programmer or compiler, depending on the thread length – e.g., a 7 or 15 register thread); or,
2. Allow threads to share registers, just as they can share memory locations. This makes independent thread migration more difficult, and to address this problem some threads will "own" the shared registers and not be permitted to migrate (thus fixing the shared registers to a node).

The former has the advantage of offering the programmer a very simple and flexible model, while the latter has the potential to allow threads to perform data synchronization without going through memory. Synchronizing on registers is particularly advantageous for threads on the same node of execution, given the results from Chapter 6 that show that very short threads perform significantly more synchronizations (more than half of which are through registers). That is, if a 10-instruction thread must perform 9 synchronizations (five of which can be encoded in registers), bypassing the memory system to perform those synchronizations would be tremendously advantageous. Furthermore, given that short threads share more data values, typically do not travel, and (because they execute few instructions) exist for a shorter amount of time than long threads, sharing registers makes sense. The one

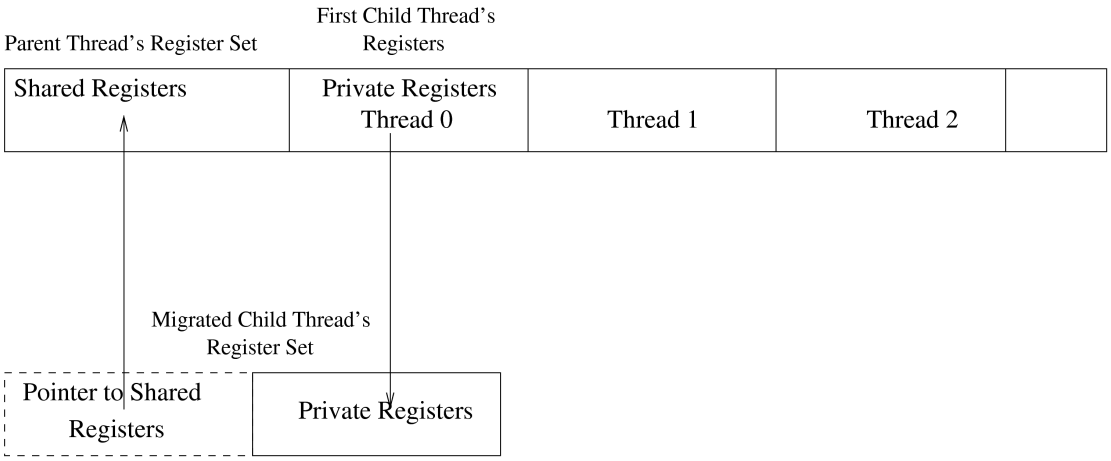


Figure 8.1. A potential Traveling Thread register configuration.

case where threads need to have minimum state is when performing a migration. Consequently, a large set of register state can be shared local to a node, but when a thread migrates only some of those registers should be transferred. And then, if the thread needs to synchronize with register state that is on another node (after the thread has migrated), there must be a mechanism for the thread to migrate back to the node where the required synchronization will occur.

Figure 8.1 shows a potential traveling thread register implementation. A single *parent* thread could be forked that allocates a full set of registers. That register set could be divided into segments: one segment that is shared among all threads, and another that is private to other *child* threads that will be forked by the parent. Parents and children could perform synchronization in the shared register area, while maintaining private and temporary state in the local register area. If parents were prevented from migrating, the register state that they allocated (for themselves and their children) would be resident on a singe node. If a child thread then migrated (or were forked as a result of a parent thread that needed a remote operation performed), that child could take its private state with it, as well as a pointer back to the

original shared state so that synchronization could be performed if necessary. This synchronization could also occur through memory, but this form of register file would allow parents and children co-resident on the same node to quickly synchronize and exchange data, while still allowing traveling child threads to synchronize with their parent. It is also analogous to register allocation policies in conventional microprocessors where, when procedures are called, some registers are reserved for "temporary" use so that they do not have to be saved to the stack (e.g., the caller is responsible for their management), and others are shared across procedure calls.

Furthermore, this parent/child register management scheme potentially allows for some flexibility in the number of registers allocated for parents and children. A long-running thread requiring more registers may consume all of them itself, while a short-running thread could pack a very small number of registers with it before moving to a remote node during a migration. Most importantly, this configuration optimizes for the most common case: a small thread that does not migrate but needs frequent synchronization.

8.1.3 Synchronization

If the 9 synchronizations in the previous example had to be performed via the memory system, two problems are introduced into the instruction stream:

1. The thread must execute additional instructions. On average, a 10 instruction thread performs 4 memory references. Assuming that each of those must be synchronized, the 5 remaining synchronizations are encoded in the dataflow graph through registers. If those had to be performed through memory, potentially 10 additional instructions (one load and one store with the appropriate synchronization for each) would have to be added to the 10 instruction thread. Merely issuing those instructions increases the thread length by 100%.
2. Unless the processor performing the synchronizations on 90% of all instructions has an infinite size, infinite speed, and infinite bandwidth memory (or a cache with a 100% hit rate), additional memory latency will be introduced into the thread by increasing the number of memory references, which significantly impacts performance.

Thus, not only should synchronizations be allowed through both registers and the memory system, they should be optimized so that they can be performed on virtually every instruction. Because short threads may have to perform a synchronization on up to 9 out of 10 instructions, most instructions should support an automatic (or implicit) thread synchronization. Every instruction that consumes a register or memory value should check to ensure that the value it is attempting to consume is available without having to add explicit synchronization. In a full/empty bit implementation, every instruction should check for "fullness" before executing. Similarly, instructions consuming a value should have the ability to reserve that value if the an update will be performed. For example, consider the incrementing of a memory value (x) via an explicit synchronization similar to the PowerPC's reservation mechanism described in Chapter 4:

```
load_and_reserve $1, x
addi $1, $1, 1
store_and_clear x, $1
```

Clearly, for a memory operation, this synchronization mechanism is adequate. Assuming some form of a `load`, `store`, and `add` instruction would have to be executed to perform the operation, adding the ability to perform the synchronization does not increase the number of instructions the thread would have to issue. The overhead is thus dictated by how long the processor takes to perform the actual synchronization. Once again, since synchronization is so prevalent, this operation must be performed quickly. It is likely that this operation will most often succeed, especially given the short life-time of threads. If this is the case, then the mechanism to restart a thread when the `load_and_reserve` instruction must wait because the data is unavailable can be slower than the original check for fullness.

Similar issues arise when performing synchronizations on registers. For single-instruction operations, some guarantee of atomicity would be sufficient. That is,

if the `addi` instruction guaranteed atomicity during its execution, and the code segment in the previous example used register `$x` instead of memory location x, the following would be sufficient synchronization:

```
addi $x, $x, 1
```

However, multi-instruction operations on registers would require a longer period of synchronization. Consider the computation of $x = y^2+1$, implemented in pseudo-assembly as follows (assuming `$x` and `$y` are in registers):

```
mul $y, $x, $x          ; square x
addi $y, $y, 1          ; increment
```

A reservation mechanism may be required for each of the source registers (in this case, `$x`), and each of the targets (in this case `$y`). There are two problems with this approach:

1. When updating a complex data structure, a larger number of registers and memory locations may need to be reserved than can conveniently be encoded in a fixed-width instruction. The example above only has two registers, but, potentially, all the registers that the thread shares may have to be reserved.
2. The number of memory locations that may need to be reserved is essentially unbounded.

Consequently, three synchronizations mechanisms are proposed: first, a register-only synchronization mechanism for very short atomic operations (e.g., the increment given above); second, a register based mechanism for reserving several of the shared registers simultaneously; and finally, a memory based mechanism for reserving data items during loads, similar to full-empty bits.

For very short, atomic operations, all instructions that consume a register will guarantee that they will not execute until all of their source registers are available. All instructions that write values to registers will further guarantee that they will not execute until the register that they are writing to is either available or has already

been reserved by the currently executing thread. For example, if registers `$1`, `$2`, and `$3` are all shared registers, the instruction `add $1, $2, $3` must guarantee that it does not execute until all the registers are available to it, and that the add occurs atomicly. Any registers that are private to the thread do not need to be reserved if threads execute in-order.

In the case of a slightly longer set of operations occurring entirely through registers, an explicit register reservation instruction is introduced:

```
reserve $a, $b, $c...
```

The `reserve` instruction takes up to the total number of shared registers as arguments and blocks the currently executing thread until they are all reserved. Naturally, this increases the number of instructions that have to be issued. However, clearing all of the reservations could be implicitly added to every instruction. Consider the simple example $x = y^2 + 1$ (assuming registers `$x` and `$y` are shared):

```
reserve $x, $y          ; reserve the required registers
mul $y, $x, $x          ; square x
addi_clear $y, $y, 1    ; increment and clear the reservation
```

The `reserve` instruction reserves all the required registers, and the combined `addi_clear` performs the last required arithmetic operation then clears the reservation on all the registers.

Finally, the memory-based mechanism for reserving items for synchronization is proposed to use two instructions:

- `load_reserve $r, $addr`: loads the value at `$addr` in memory into register `$r`, and reserves memory address `$addr`.
- `store_clear $addr, $r`: stores the value in register `$r` to memory address `$addr` and clears any reservation.

As with any other instruction, registers `$r` and `$addr` must be available (e.g., not already reserved by another thread). Unlike the PowerPC reservation mechanism,

the `load_reserve` instruction will block the thread until the memory address `$addr` is unavailable for a load; similarly, `store_clear` will resume any threads blocked on `$addr` when the store occurs. This eliminates the test required by the PowerPC's mechanism.[1]

It should be noted that these synchronization mechanisms can be implemented in a number of ways, including using full/empty bits, or via table lookups performed by the processor (in the case of memory synchronizations these tables would have to be associative). The implementation could also occur at various levels of granularity: a byte, a machine word, or a larger size (e.g., a cache line, or an open row from memory).

Finally, complex synchronizations will be plagued by the potential for deadlock, and these synchronization mechanisms provide no specific means for avoiding it. The programmer or compiler will have to employ a safe locking protocol for deadlock avoidance.

8.1.4 Migration

The migration mechanisms remain the same as those enumerated in Section 4.2.2. Since, in this scheme, parent threads cannot migrate (because they may "own" registers that my be required by their children), it is proposed that any non-local memory reference by a parent thread generate an exception (as described previously in Section 4.2), and that the parent thread can handle the exception by optionally forking a child to perform the remote memory reference. In the case of child threads, any remote memory reference will generate an automatic migration that packages the child thread's private registers along with a pointer back to the parent thread's

[1] It may also be advantageous to include polling versions of these instructions so that each thread has control over when it may block, however this is an optimization trade-off between the time it takes to poll, the time it takes for a thread to go to sleep and to wake up, and the probability of the thread blocking for a "long" time.

register set (so that global registers may be referenced). Consequently, children should attempt to maintain the data they need to perform their computation in private registers if they are likely to migrate. This implicit migration mechanism saves the thread having to check whether or not its loads and stores succeeded after every one.

Finally, for completeness, an explicit `migrate $addr` instruction will be provided that forces the thread to move to the node containing the address `$addr`.

8.2 Programming

Programming a machine such as this is the next big challenge. Beyond whatever the compiler can do automatically to manage threads and synchronization, additional mechanisms for the programmer to expose three key elements must be enhanced in future programming languages:

1. **Data Placement**: programming languages must provide the ability for the programmer to describe the locality of interrelated data structures.
2. **Concurrency**: additional methods need to be provided for the programmer to specify concurrency (e.g., special looping structures that result in many new threads being created).
3. **Synchronization**: the programmer must be able to specify which data items need to be synchronized, and potentially how at a high level. In particular, an explicit mechanism of specifying all of a thread's producer/consumer interactions would significantly simplify programming (especially for longer running parent threads).

Proposals in each of these areas have been made in the parallel computing community, but generally not for architectures with as fine grain support for short threads as the one proposed in this chapter. Coarray and High Performance Fortran have both proposed methods for dividing data among parallel nodes and describing which data should be coresident. Most conventional multiprocessors support programmer defined threads, as well as MPI for explicitly describing concurrency. In

fact, on MPPs, MPI tends to do a good job of describing the physical structure represented by the parallel application. And finally, the Cray MTA proposed a *futures* mechanism for describing synchronization.

Each of these mechanisms is a good beginning. For example, MPI can often describe ten-thousand way concurrency in many applications. To scale to trans-petaflop size, with perhaps two orders of magnitude more nodes, additional mechanisms for describing concurrency must be employed. However, this can happen below the level of MPI (e.g., there is no need to discard the concurrency already described by the programmer).

In the case of data placement, understanding the relationship between static arrays is important, but mechanisms for describing *where* a dynamically allocated data item should reside simply do not exist today. Furthermore, it is extremely difficult (if not impossible) to describe a virtual address to physical address mapping that allows contiguously allocated objects in virtual address space (e.g., arrays) to be mapped to physical locations throughout the system that are coresident with other objects in the virtual address space (e.g., other arrays) on most architectures.

Synchronization may be the most difficult issue, given how pervasive it is. The compiler must find efficient, deadlock free methods of synchronizing between threads automatically, which is a rich and open topic of research.

8.3 Conclusions

The results of this dissertation represent nearly one compute year worth of simulation effort, over 100 thousand lines of code, and generated nearly a terabyte of intermediate results.

Chapter 3 demonstrated that the integer and floating point benchmarks used throughout this dissertation require significantly more from the memory system than

the dominant computer architecture benchmarks (the SPEC suite). Over the same number of instructions, the Sandia codes consume, on average, 4 times the unique memory bandwidth of their SPEC counterparts. In the critical L1 cache size region, the temporal working set of the Sandia suite averages a nearly 25% increase in miss rate. For the floating point codes, the Sandia applications perform over 5.5 times the number of integer operations of their SPEC counterparts, indicating a much more complex and configurable set of applications. The resulting difference in memory requirements can result in a 70%-80% degradation in application performance.

The experimentation has demonstrated that the traveling thread execution model has the potential to increase concurrency within today's applications, on average, by 1-2 orders of magnitude (Chapter 5), with some sections of the code potentially capable of issuing 5-6 orders of magnitude more instructions than can be exploited by conventional processors. Based on the critical path through the dataflow graphs, a 30% to 60% reduction in execution time is possible simply by executing more in parallel than can be done today.

The results show that instructions consume a small number of unique values (on average, 1.6, as opposed to the maximum of 3 permitted by the instruction set architecture studied). The values produced by instructions are consumed very, very quickly (in the median case, on the next instruction). This indicates that small threads could be created to take advantage of the relatively local synchronization required by the instruction stream.

Chapter 6 demonstrated that the instruction stream could be broken into small threads, but that the cost of doing so is an increase in the relative number of synchronizations. Specifically, on average, a 10 instruction thread will require an average of 9 synchronizations, as opposed to a 100 thousand instruction thread which only requires an average of 10 thousand synchronizations. This is, in fact, the cost

of creating many small threads. The benefits are the increased concurrency (leading to shorter execution times), the reduction of individual state size, and a nearly 60% reduction in thread migrations. In fact, the vast majority of synchronizations for short threads are local synchronizations (e.g., between two threads on the same node of execution), that can be done much faster than remote synchronizations (because they require high latency remote communication).

Furthermore, Chapter 7 shows that improvements in data partitioning techniques have the potential to reduce traveling thread migrations by over an order of magnitude.

When compared to a very optimistically constructed conventional machines that assumes a perfect ability to issue instructions out of order and perfect branch prediction traveling threads execute the program significantly faster. In fact, a conventional architecture would have to allow up to 40× the number of simultaneous transactions on its memory bus than can be performed today in order to execute the program in equivalent time to a traveling thread, which represents a massive departure from conventional designs.

The results also demonstrate that traveling threads benefit from shorter thread lengths which decrease the probability that the thread will migrate (hence decreasing high-latency network transactions), and decrease the amount of state required for each thread, all while increasing concurrency. The only cost for the benefit of fewer migrations, smaller state size, and increased concurrency is the introduction of overhead from increased requirements for inter-thread synchronization. The increase in concurrency and decrease in network transactions and bandwidth requirements greatly outweighs the additional cost of synchronization, most of which occur locally rather than remotely.

Finally, the traveling thread model can fundamentally simplify important pieces of the architecture (e.g., by replacing an out-of-order execution unit with a simpler in-order multithreaded execution mechanism). This lowers the chip area required to implement any computer, which dramatically impacts the cost of the chip by increasing yield. In the case of PIM systems, reducing the area of the processor on the chip has a two-fold impact: first, it lowers cost because the total chip area can be shrunk, or more of the chip area can be devoted to memory (which can be more easily repaired if there is a manufacturing problem with the chip, increasing yield); and second, in the case where the two chips are equivalent size but one contains more memory, the additional state is used to improve performance by increasing spatial locality and decreasing the number of thread migrations.

8.4 Future Work

Programming language designers and compiler writers have significant opportunity for research in extracting the type of concurrency described in this dissertation. The programming language infrastructure to support this execution model must be developed, as well as methods for specifying the which data items must reside together. This dissertation describes the type of synchronization mechanism to be implemented, but not its specific implementation, which must be fixed before the full cost of synchronization can be computed. And finally, mapping all of these structures to a Processing-In-Memory architecture that takes advantage of the low-latency, high-bandwidth memory available on the chip offers significant opportunities for future work in both computer and system architecture.

APPENDIX A

FULL BENCHMARK CHARACTERISTIC DATA

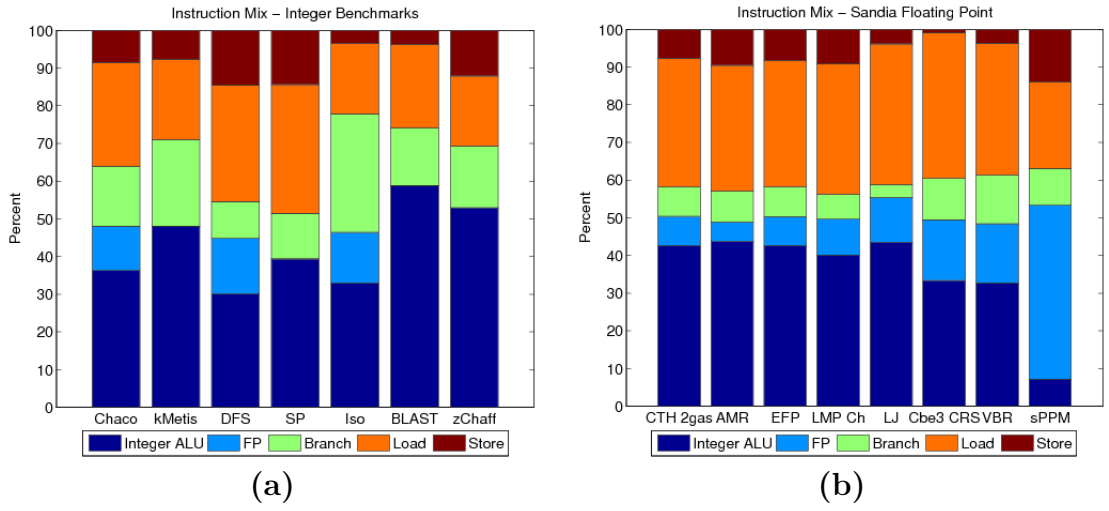


Figure A.1. Individual Benchmark Instruction Mixes for the Sandia Integer (a), Sandia Floating Point (b), SPEC Integer (c,d), and SPEC Floating Point (e) Suites. (Continued on the next page.)

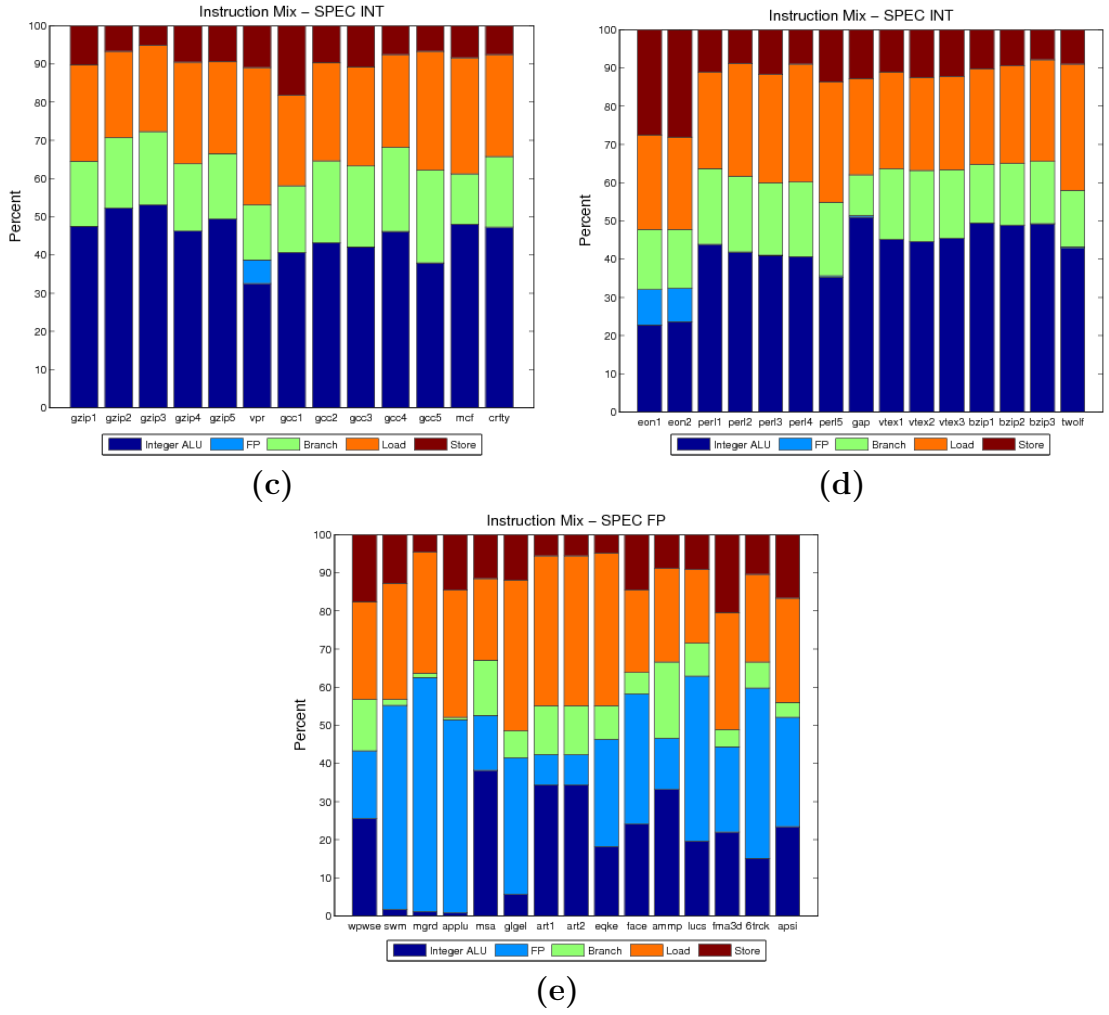


Figure A.1 (continued) Individual Benchmark Instruction Mixes for the Sandia Integer (a), Sandia Floating Point (b), SPEC Integer (c,d), and SPEC Floating Point (e) Suites.

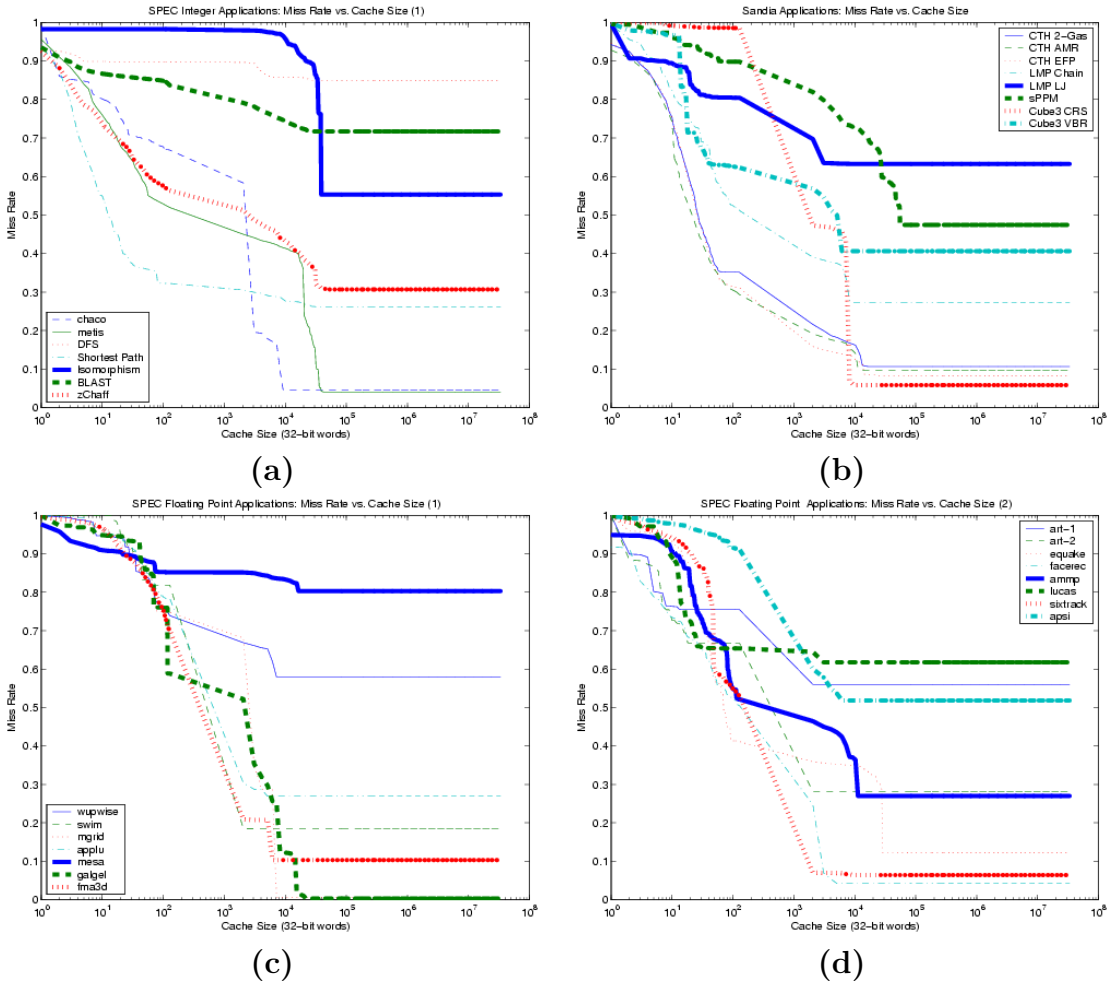


Figure A.2. Individual Benchmark Working Set Miss Rates for the Sandia Integer (a), Sandia Floating Point (b), SPEC Integer (c,d), and SPEC Floating Point (e-h) Suites. (Continued on the next page.)

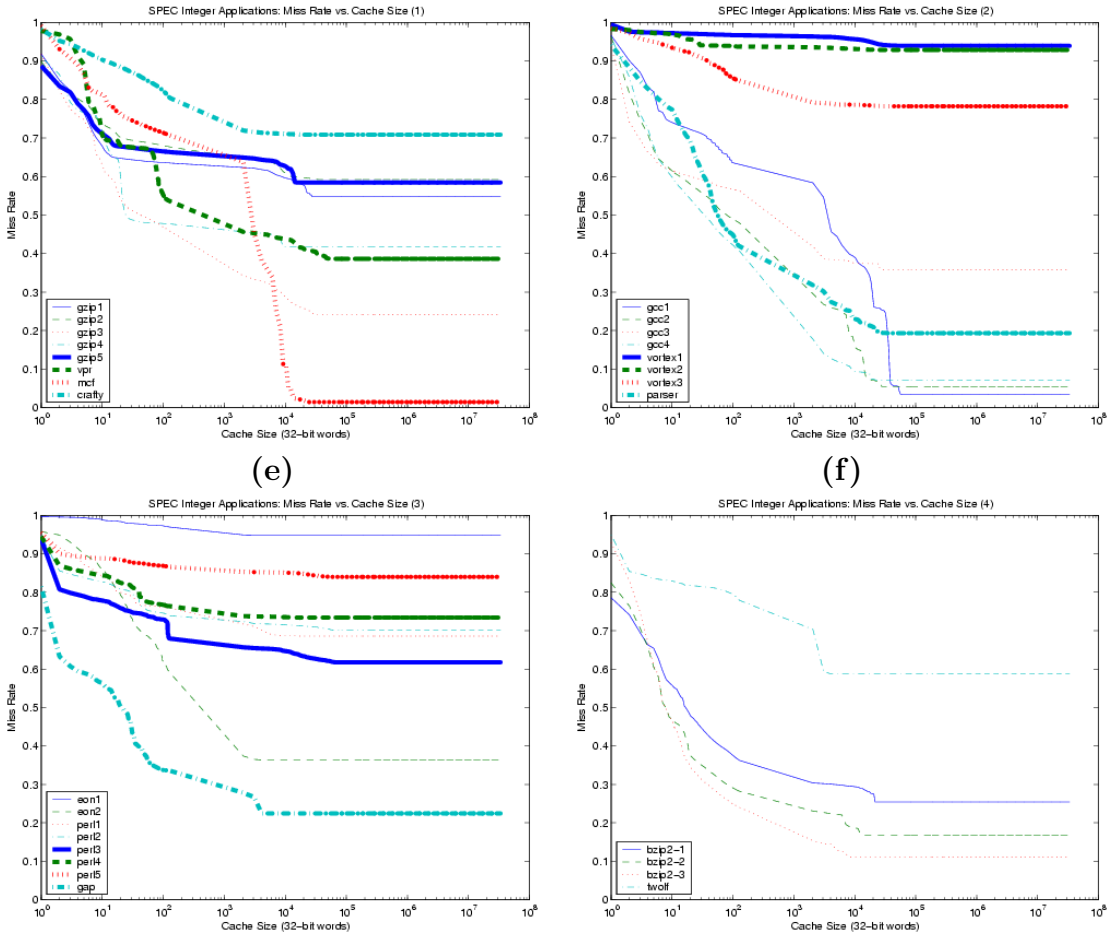


Figure A.2 (continued) Individual Benchmark Working Set Miss Rates for the Sandia Integer (a), Sandia Floating Point (b), SPEC Integer (c,d), and SPEC Floating Point (e-h) Suites.

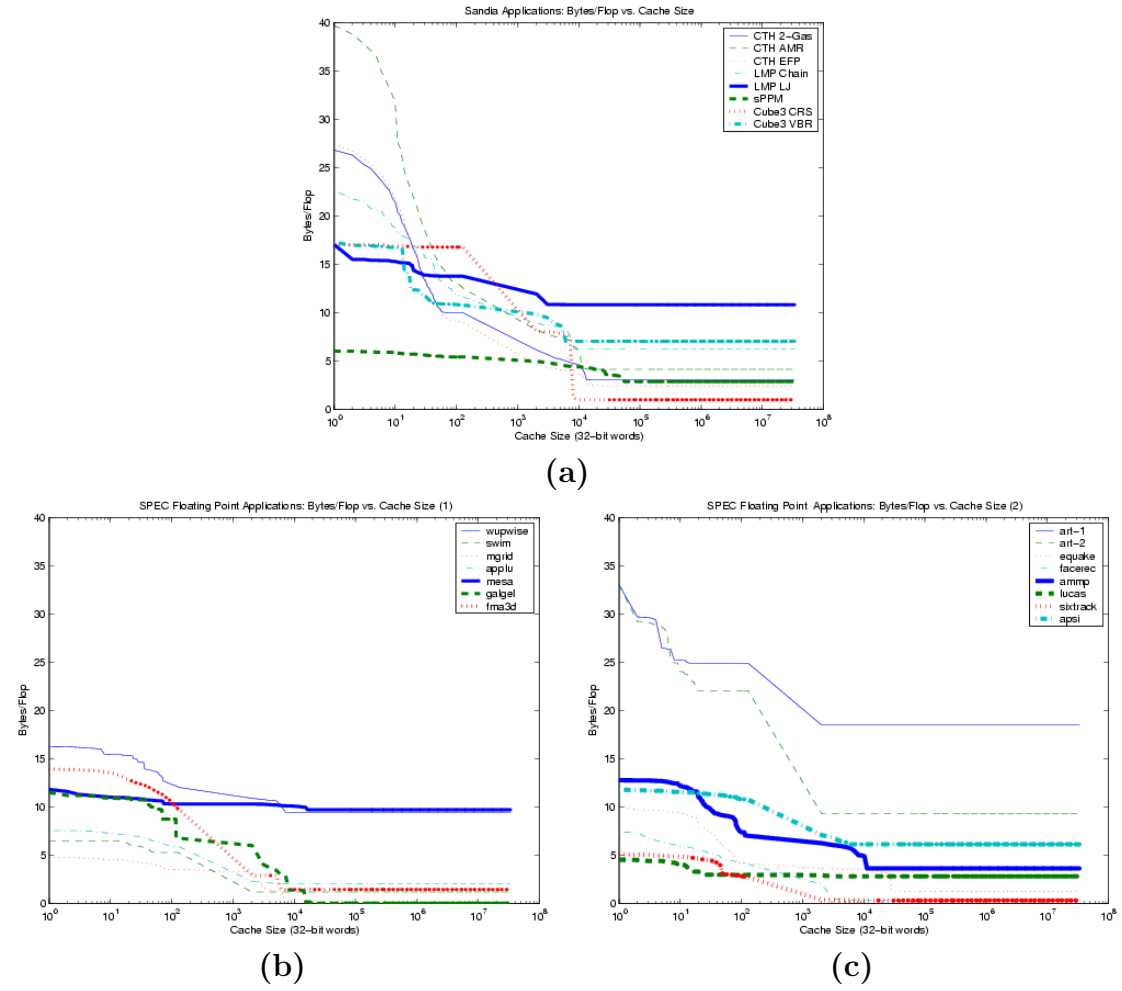


Figure A.3. Individual Benchmark Working Set Bytes/Flop for the Sandia (a) and SPEC-FP (b,c) suites

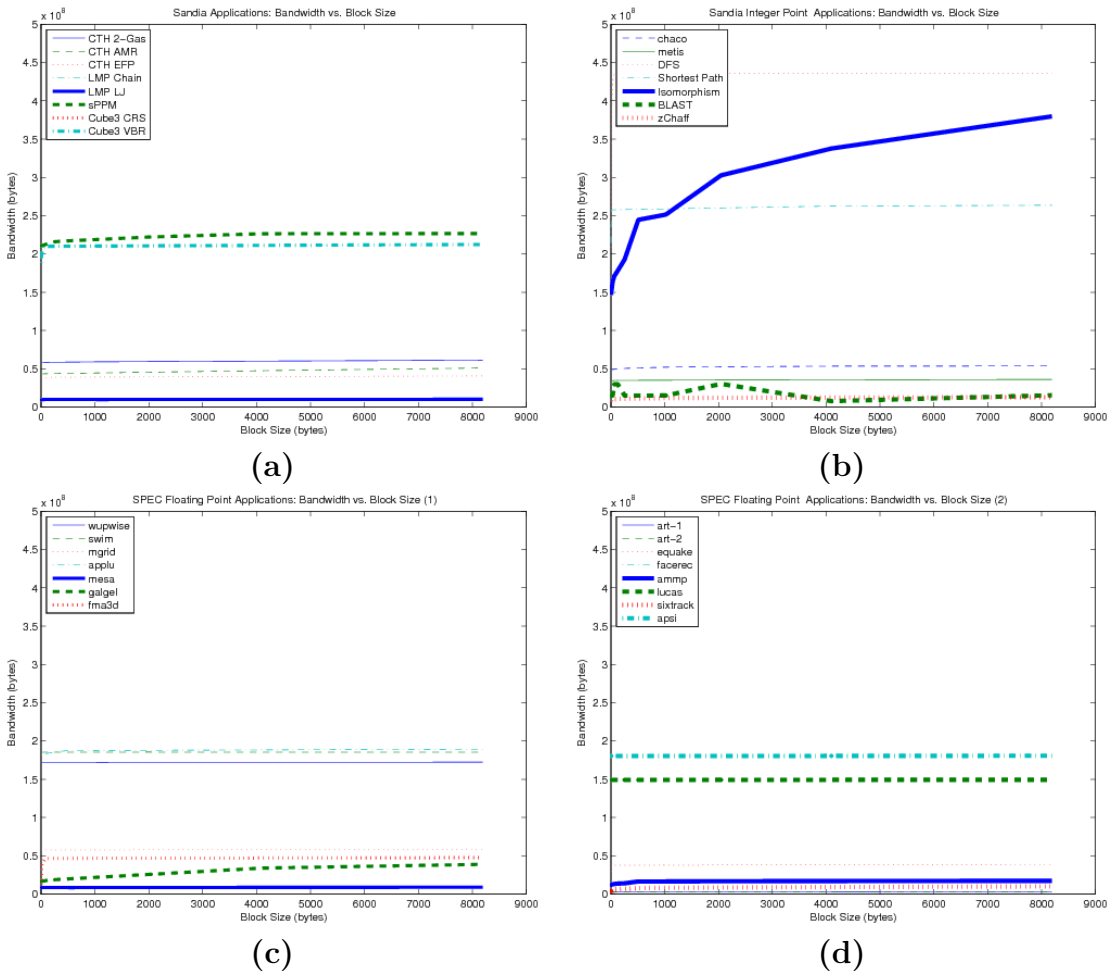


Figure A.4. Individual Benchmark Spatial Locality Results for the Sandia Integer (a), Sandia Floating Point (b), SPEC Integer (c,d), and SPEC Floating Point (e-h) Suites. (Continued on the next page.)

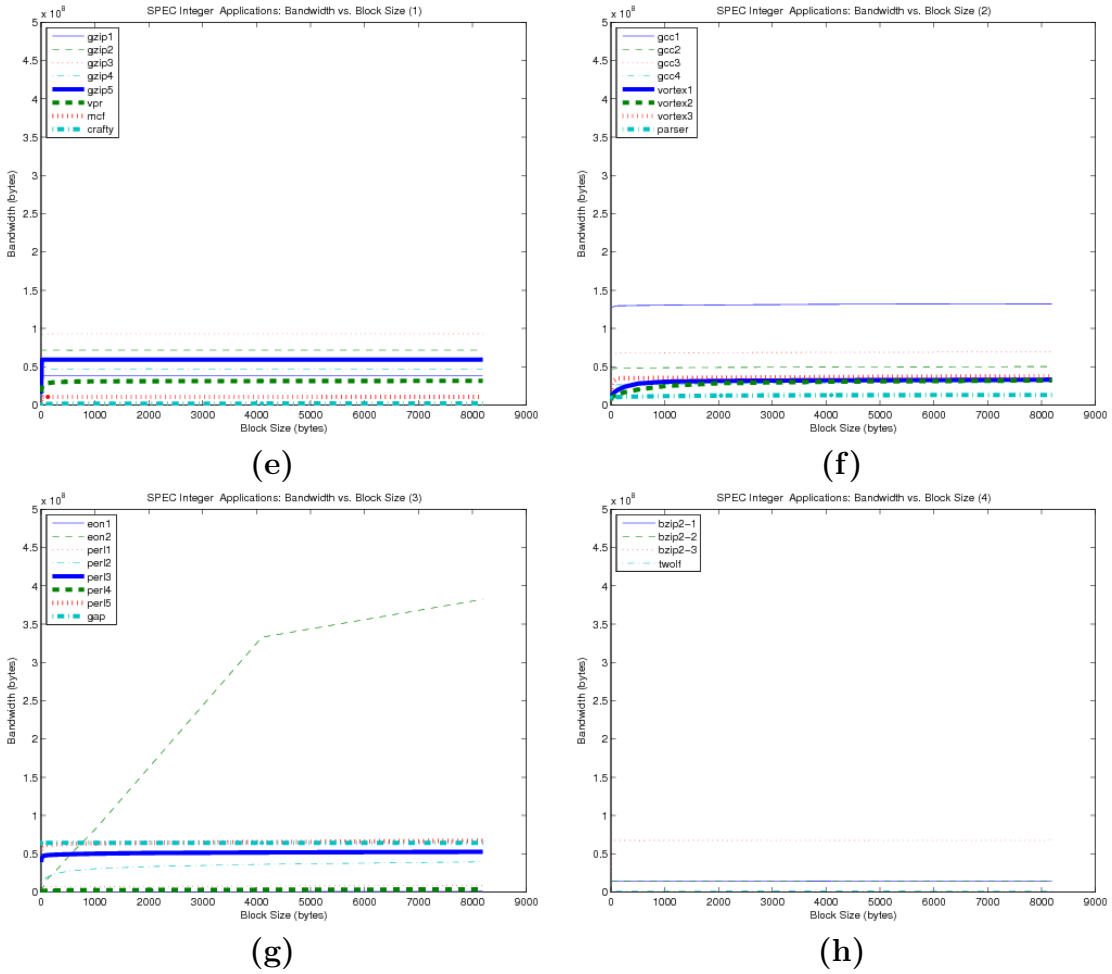


Figure A.4 (continued) Individual Benchmark Spatial Locality Results for the Sandia Integer (a), Sandia Floating Point (b), SPEC Integer (c,d), and SPEC Floating Point (e-h) Suites.

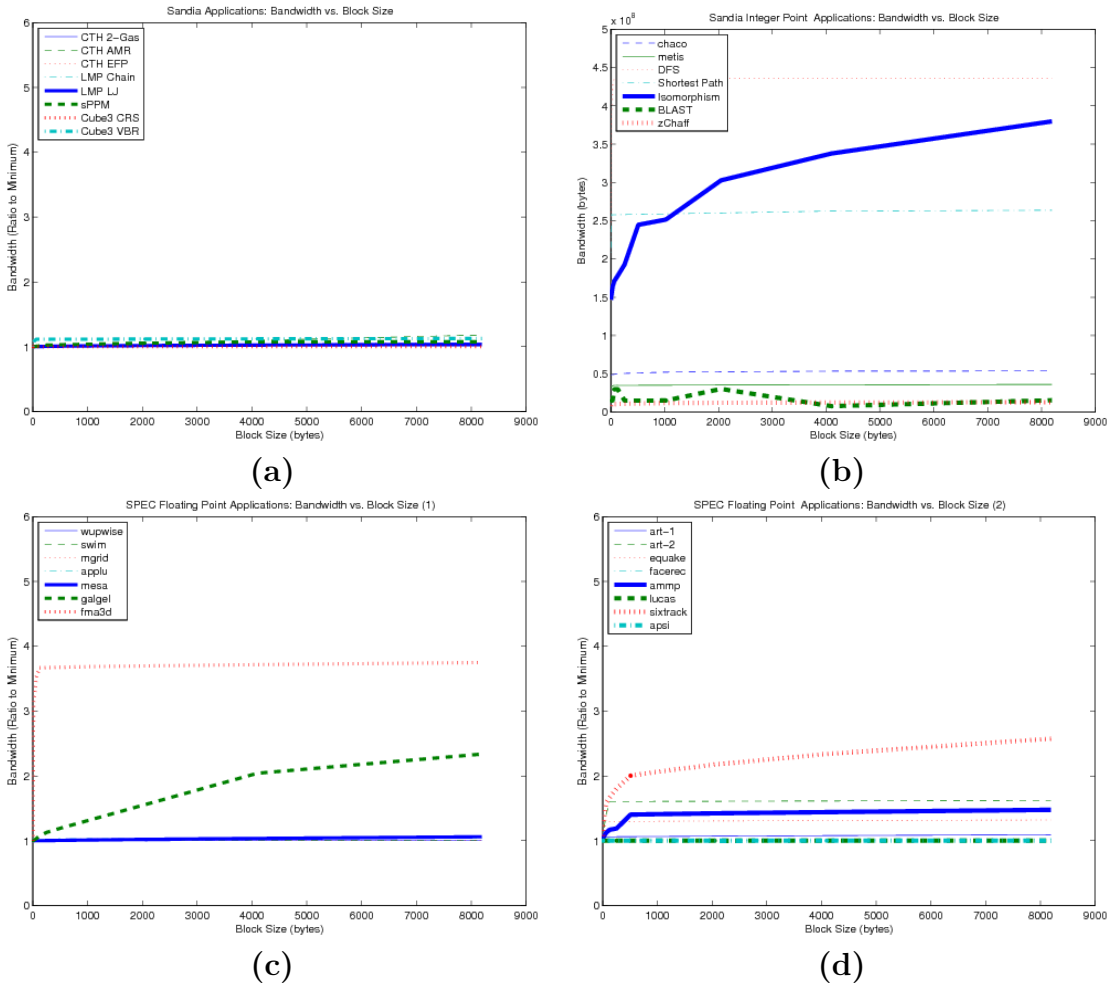


Figure A.5. Individual Benchmark Spatial Locality Overhead Results for the Sandia Integer (a), Sandia Floating Point (b), SPEC Integer (c,d), and SPEC Floating Point (e-h) Suites. (Continued on the next page.)

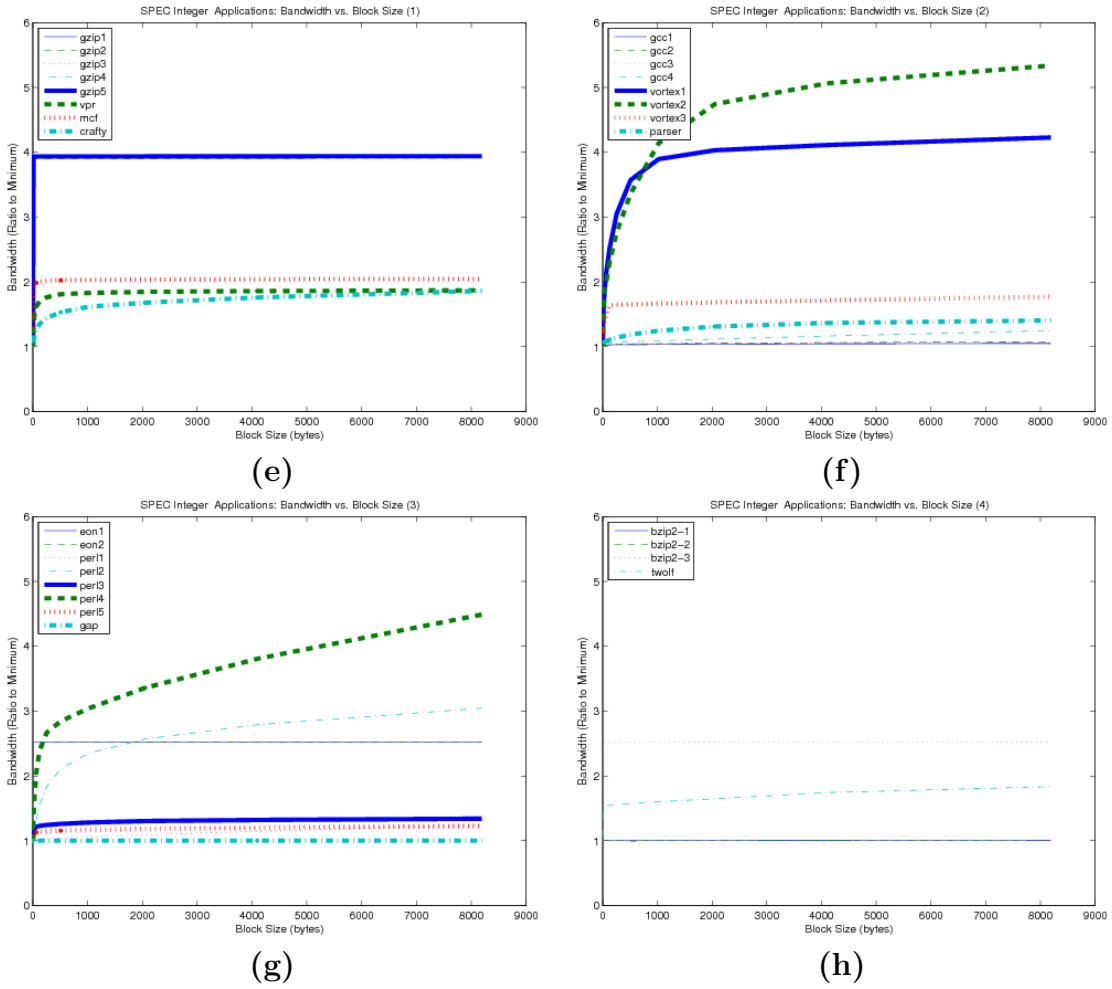


Figure A.5 (continued) Individual Benchmark Spatial Locality Overhead Results for the Sandia Integer (a), Sandia Floating Point (b), SPEC Integer (c,d), and SPEC Floating Point (e-h) Suites.

APPENDIX B

FULL DATAFLOW RESULTS

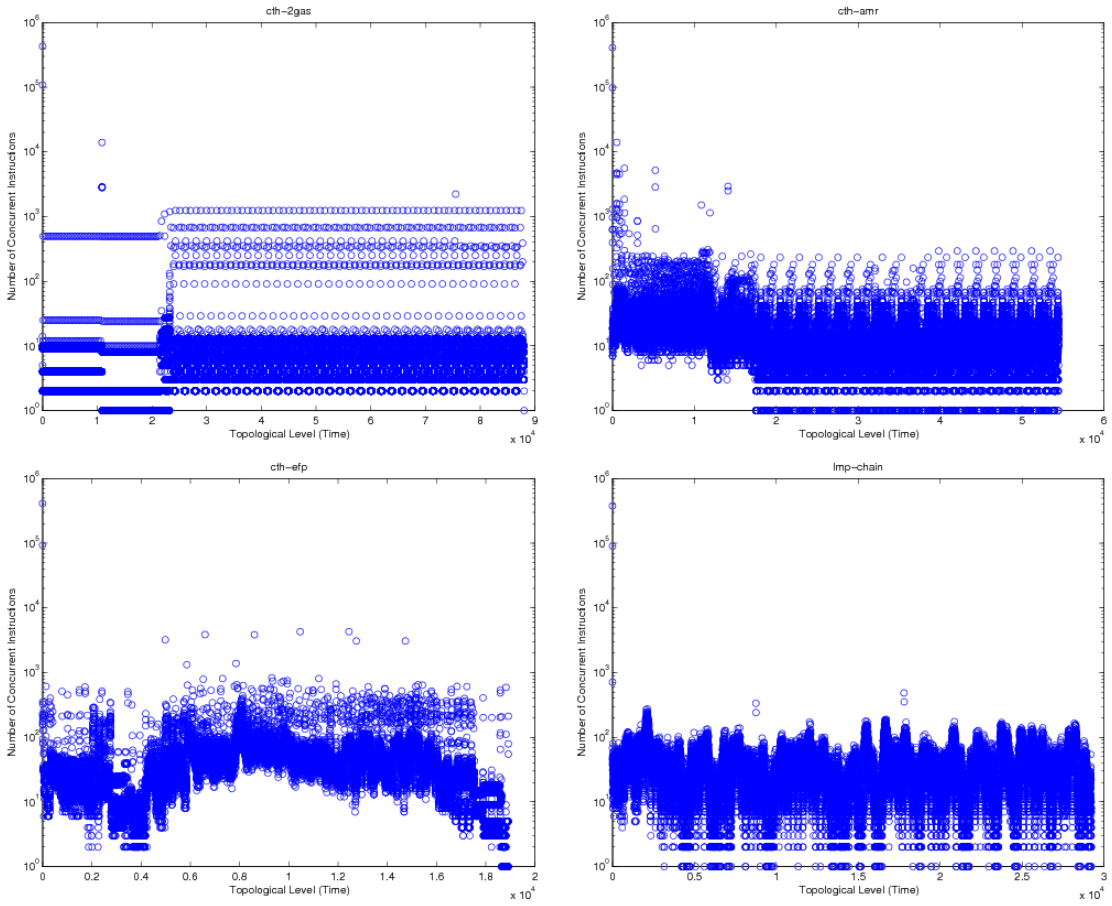


Figure B.1. Topological Layering for the Sandia Floating Point Benchmark Suite. (Continued on the next page).

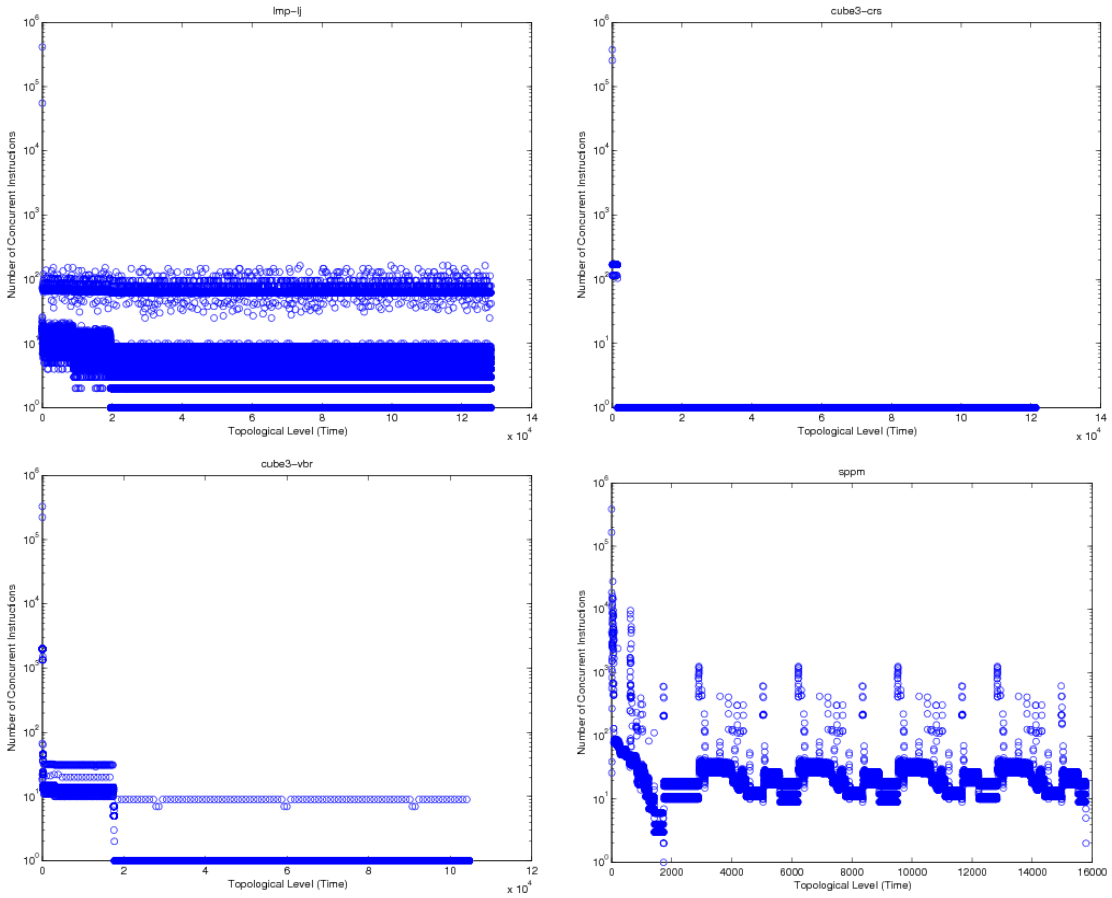


Figure B.1 (continued) Topological Layering for the Sandia Floating Point Benchmark Suite.

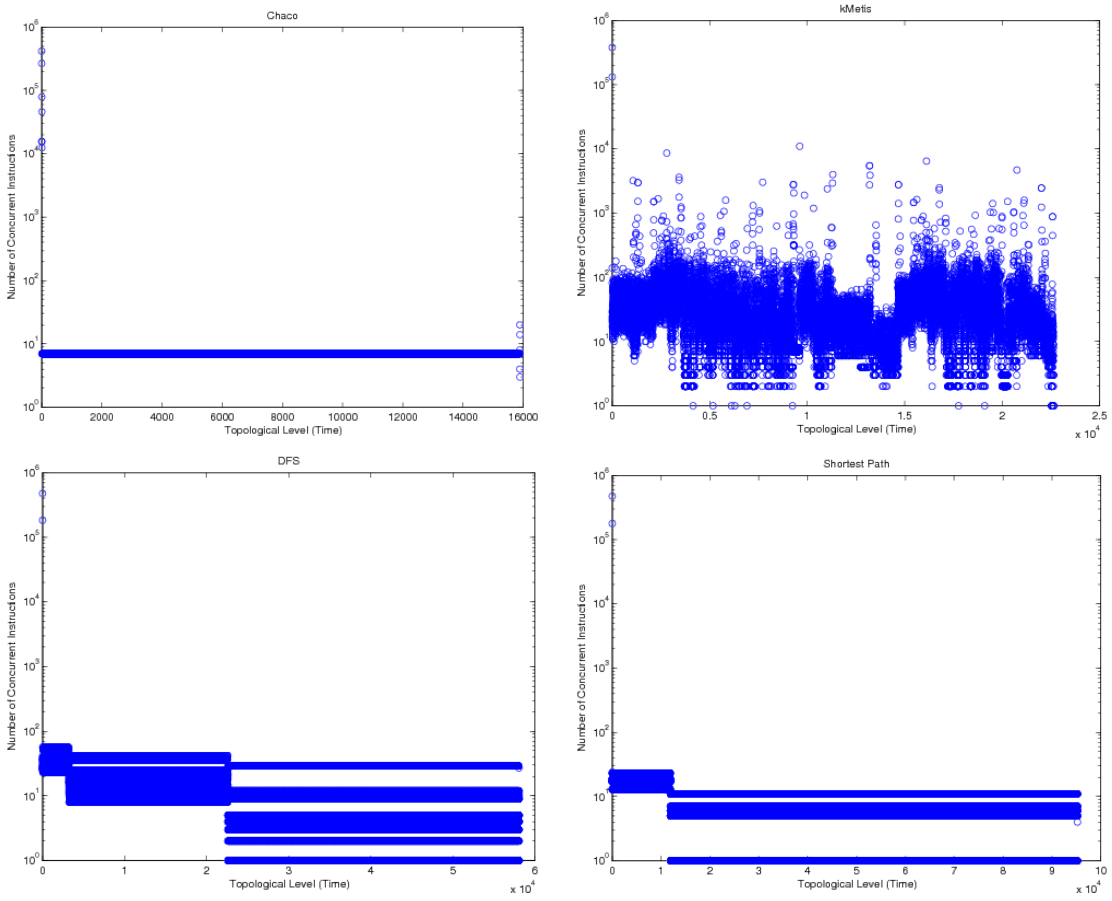


Figure B.2. Topological Layering for the Sandia Integer Benchmark Suite. (Continued on the next page.)

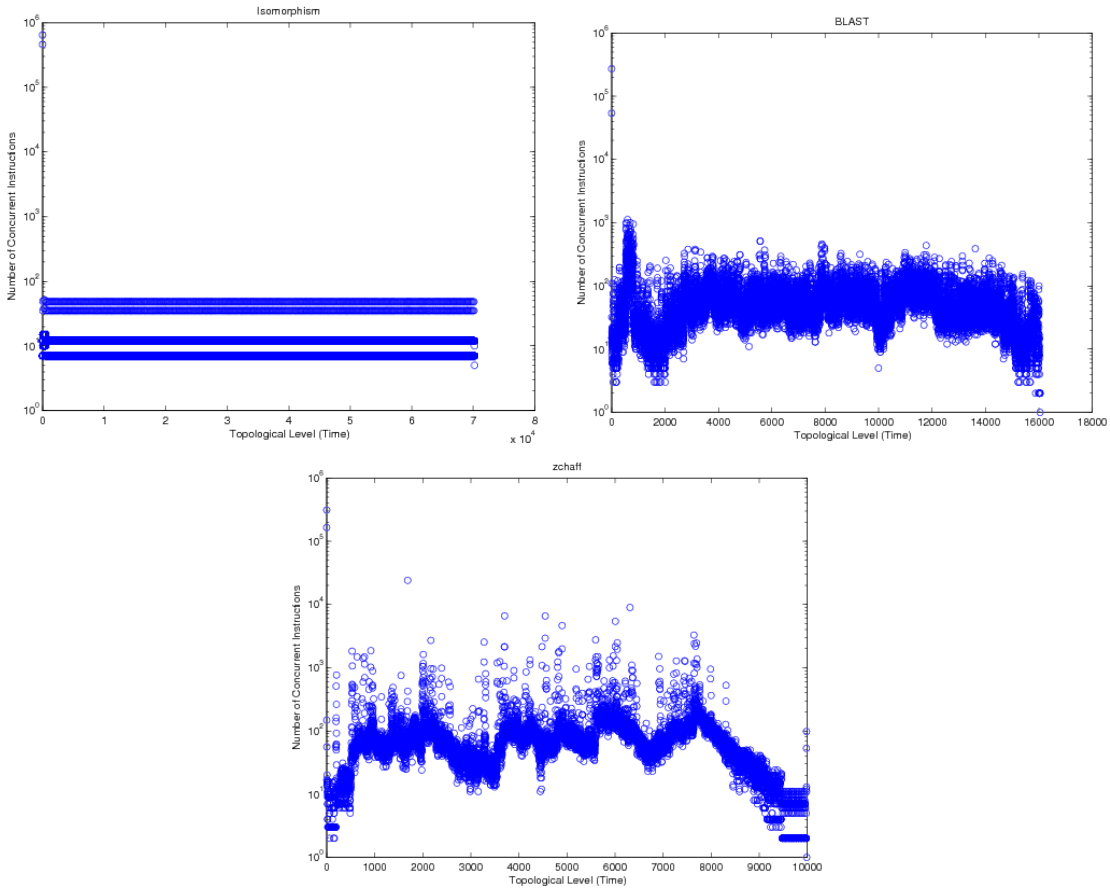


Figure B.2 (continued) Topological Layering for the Sandia Integer Benchmark Suite.

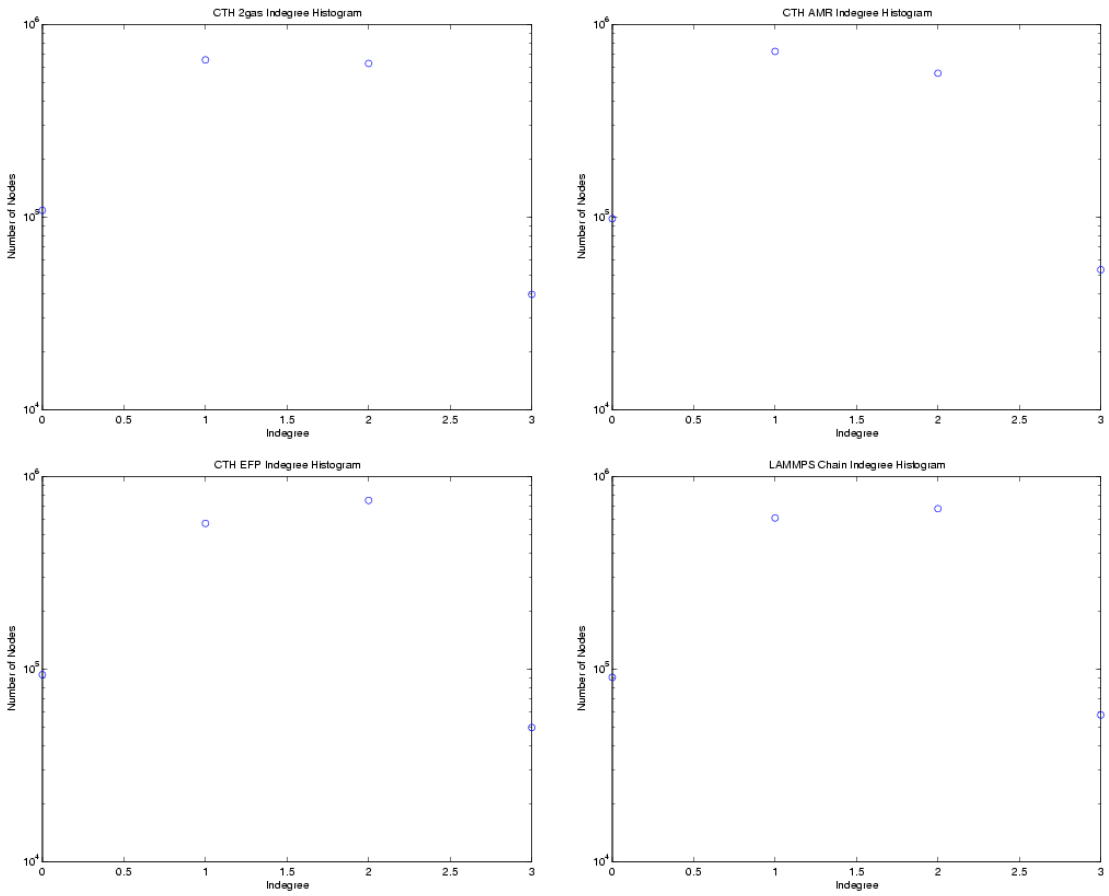


Figure B.3. Sandia Floating Point Suite Indegree Histogram. (Continued on the next page.)

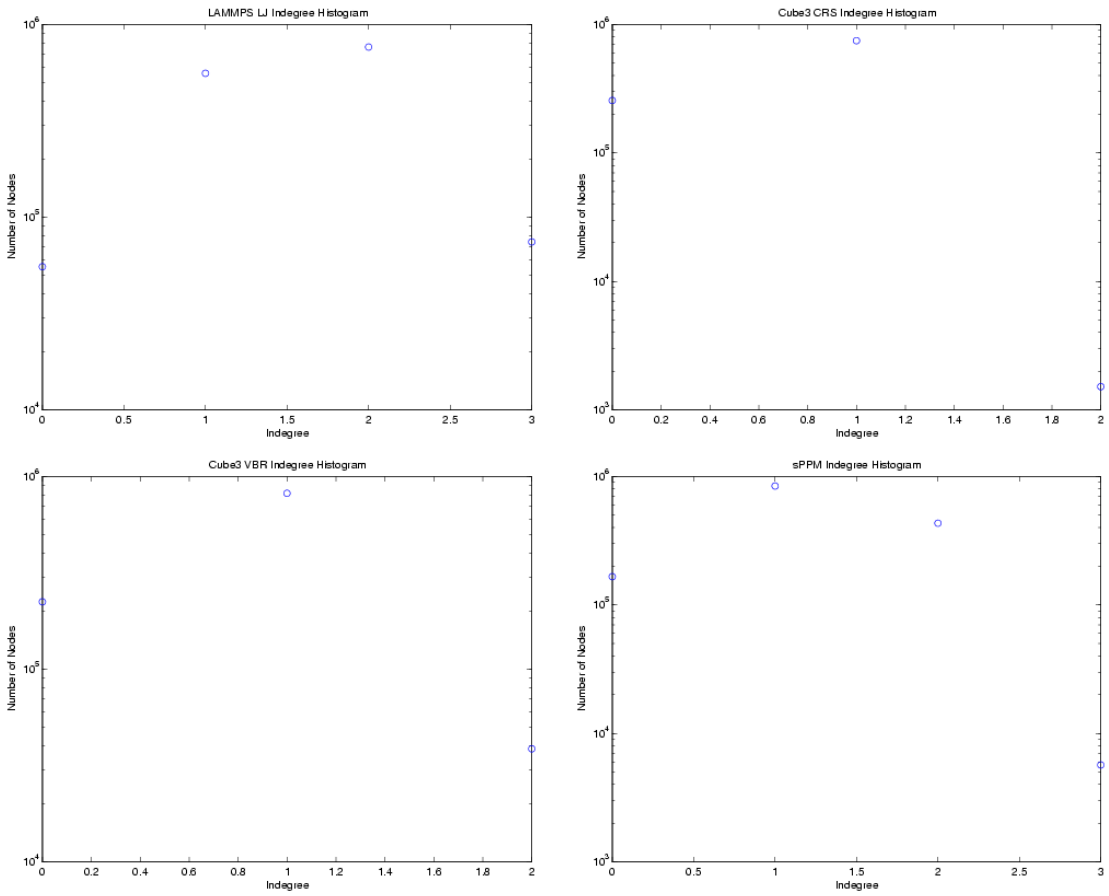


Figure B.3 (continued) Sandia Floating Point Suite Indegree Histogram.

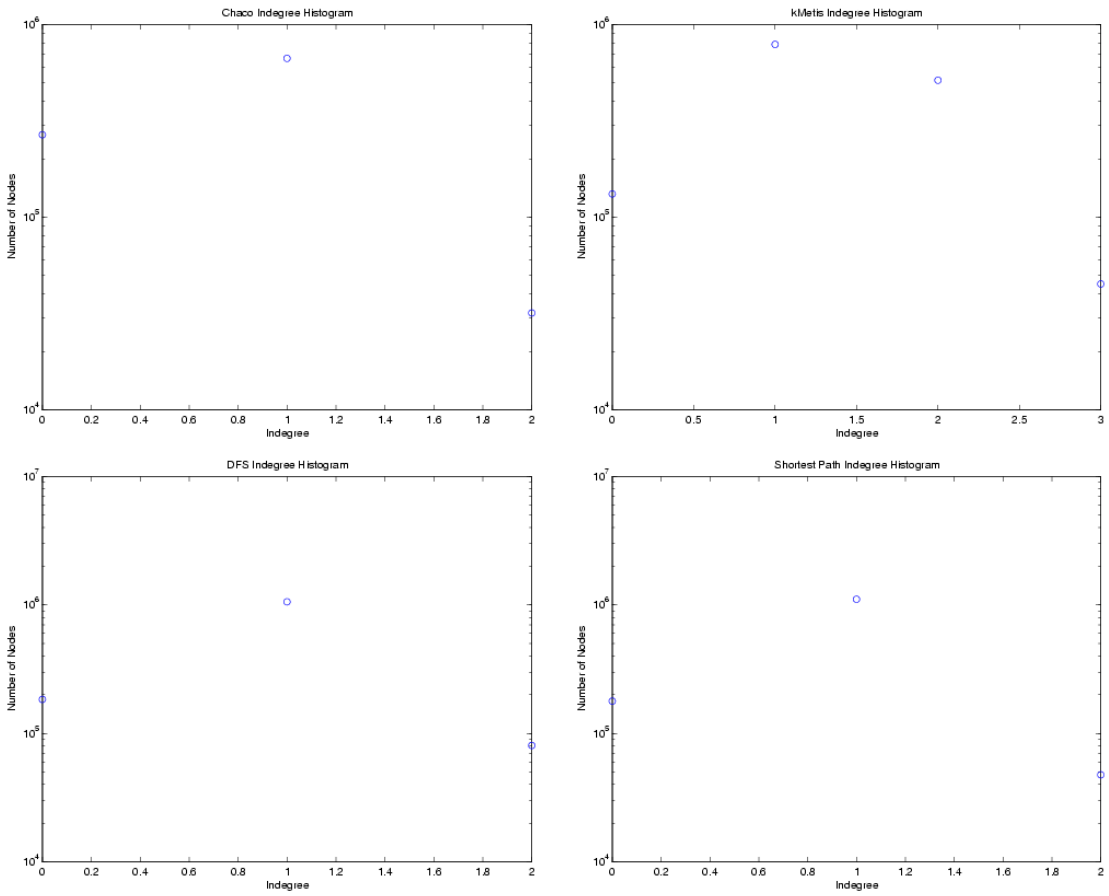


Figure B.4. Sandia Integer Suite Indegree Histogram. (Continued on the next page.)

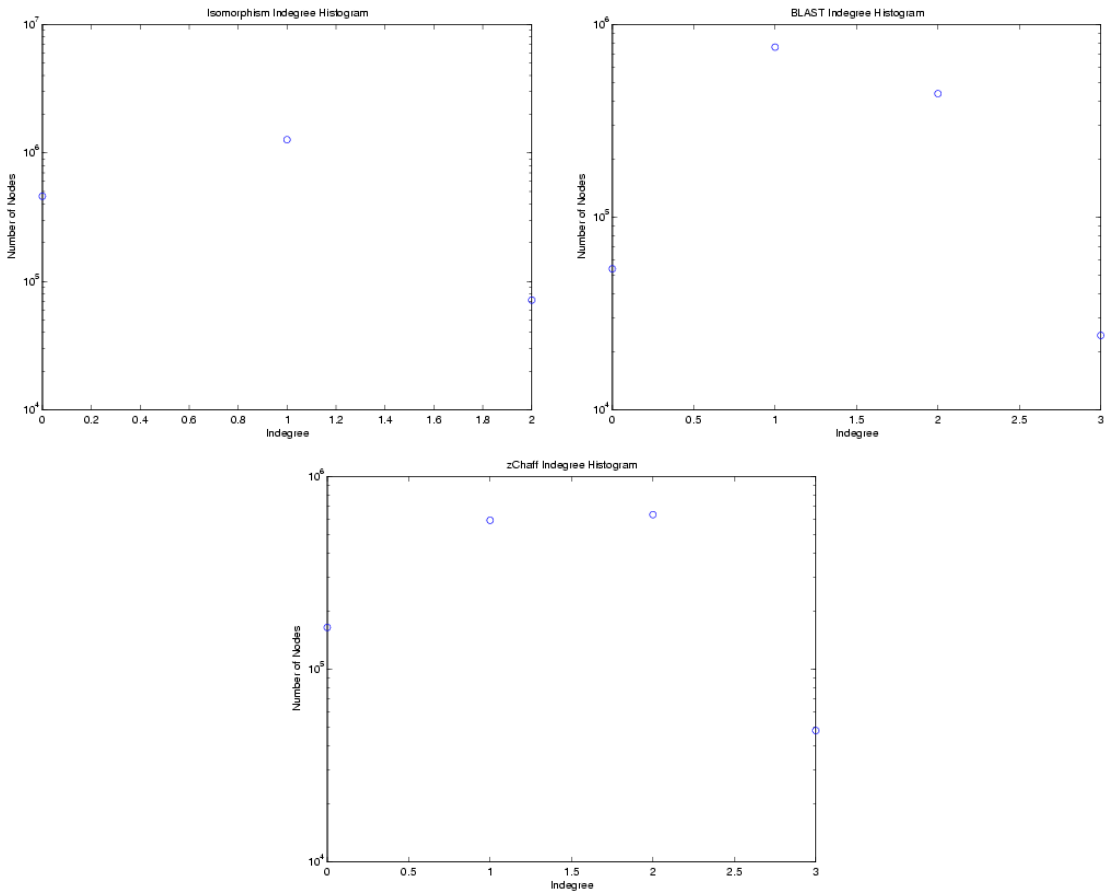


Figure B.4 (continued) Sandia Integer Suite Indegree Histogram.

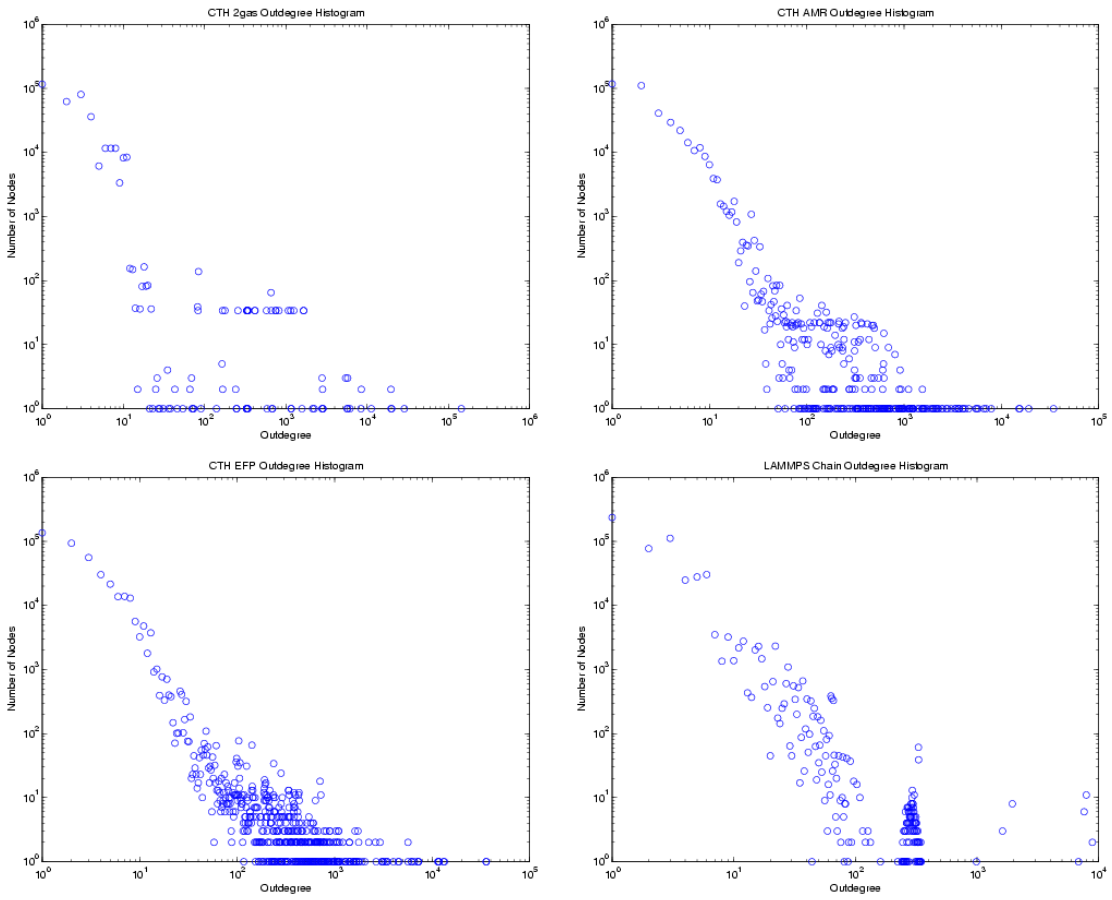


Figure B.5. Sandia Floating Point Suite Outdegree Histogram. (Continued on the next page.)

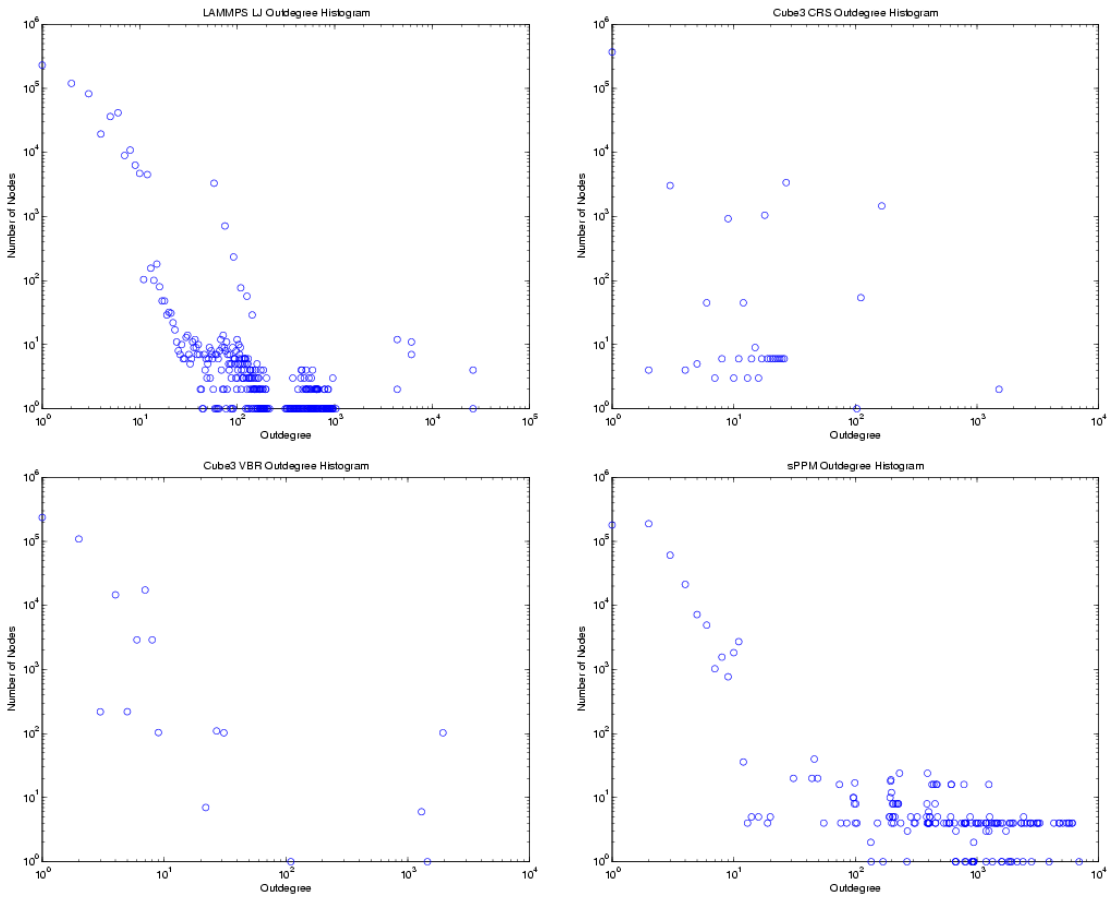


Figure B.5 (continued) Sandia Floating Point Suite Outdegree Histogram.

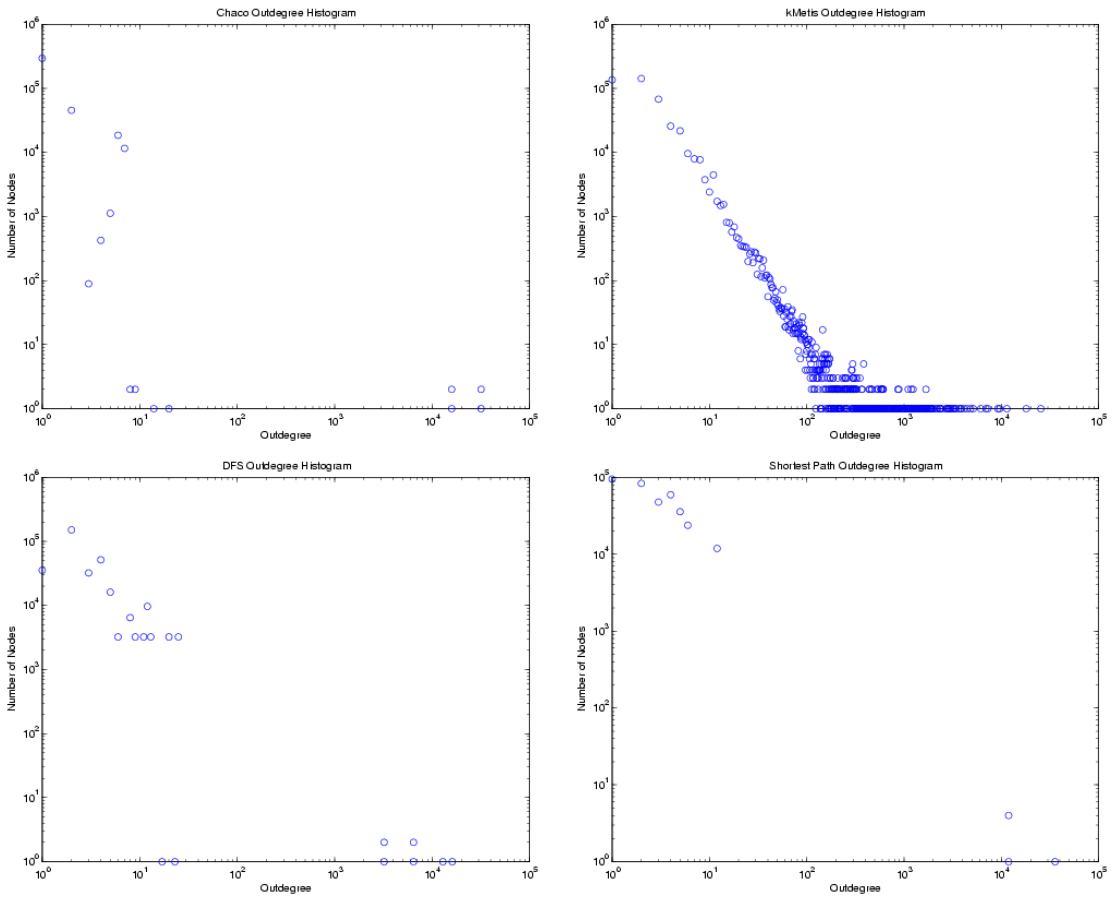


Figure B.6. Sandia Integer Suite Outdegree Histogram. (Continued on the next page.)

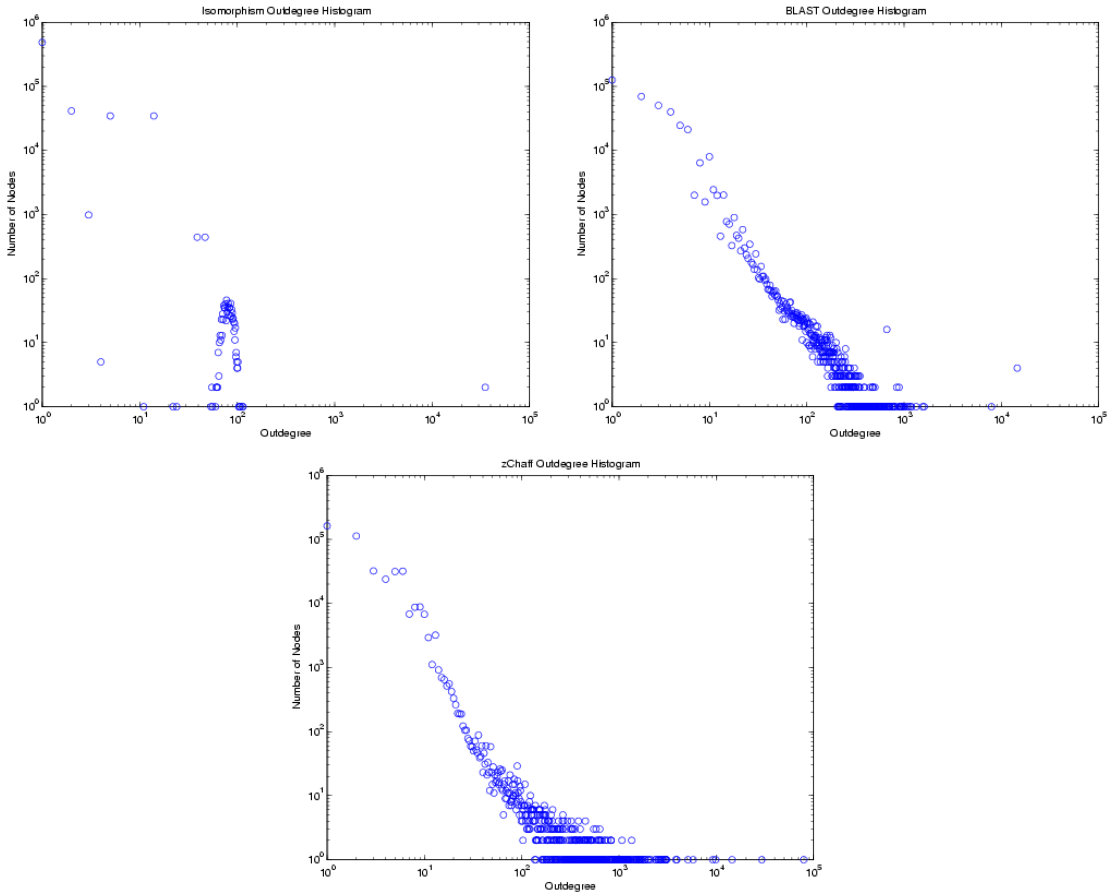


Figure B.6 (continued) Sandia Integer Suite Outdegree Histogram.

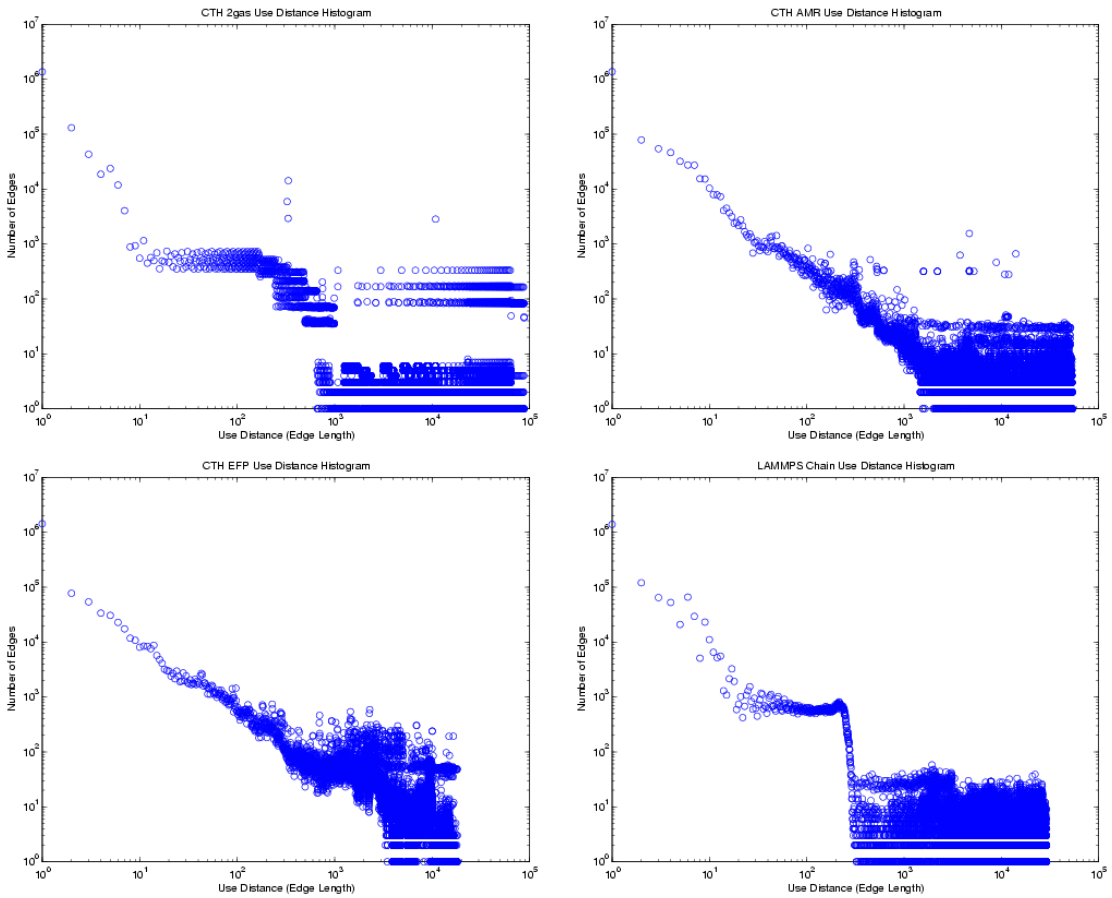


Figure B.7. Sandia Floating Point Suite Use Distance Histogram. (Continued on the next page.)

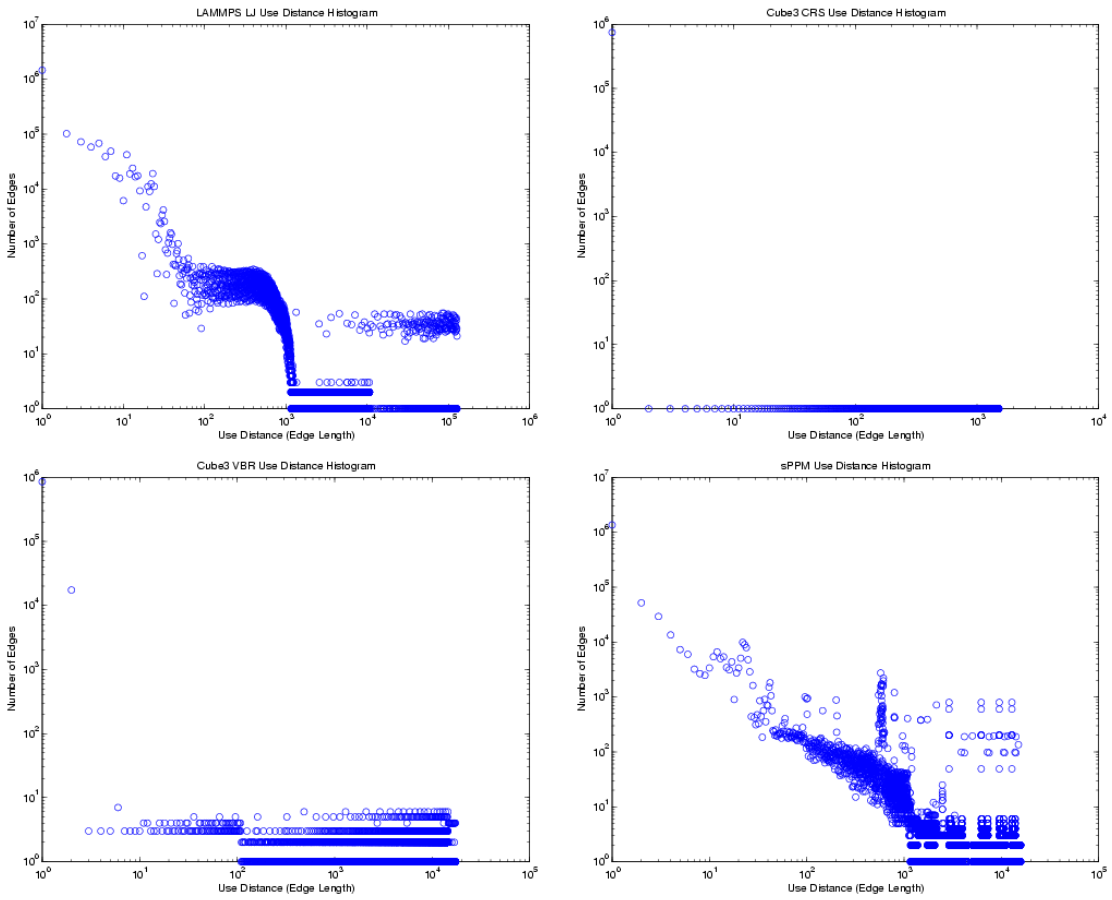


Figure B.7 (continued) Sandia Floating Point Suite Use Distance Histogram.

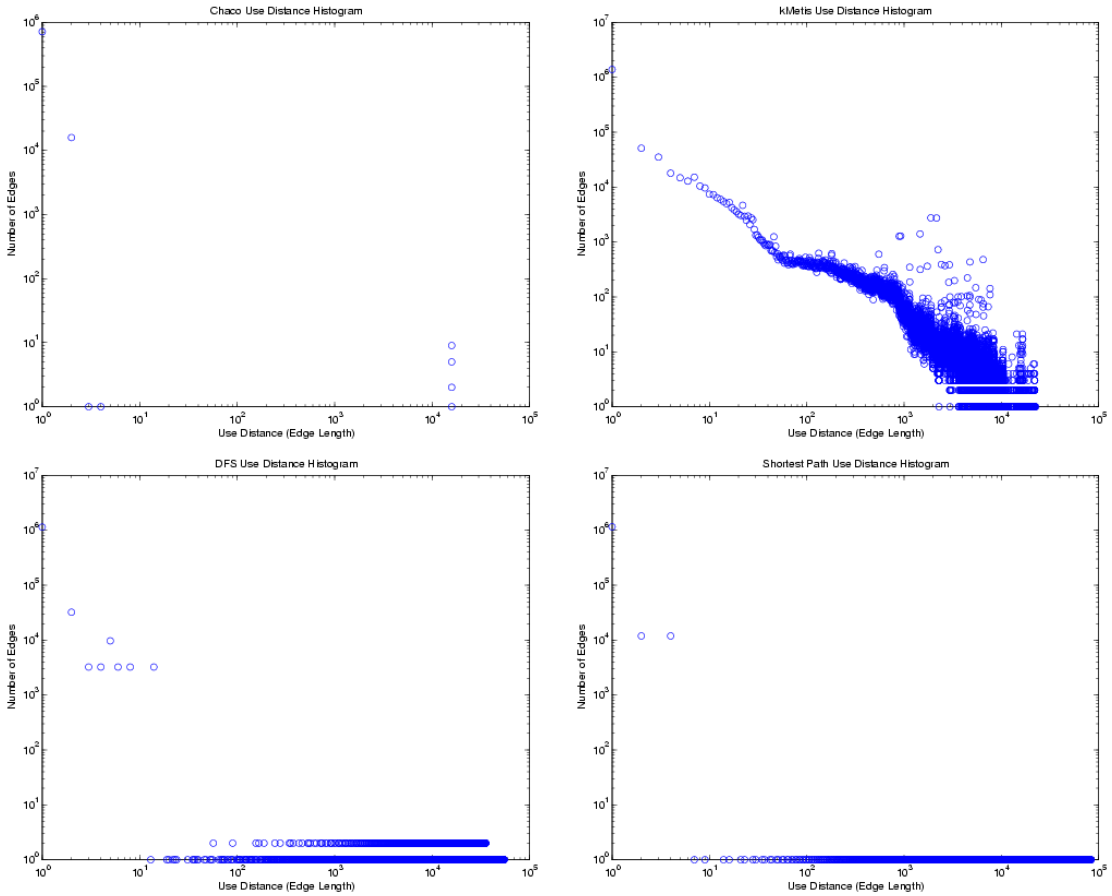


Figure B.8. Sandia Integer Suite Use Distance Histogram. (Continued on the next page.)

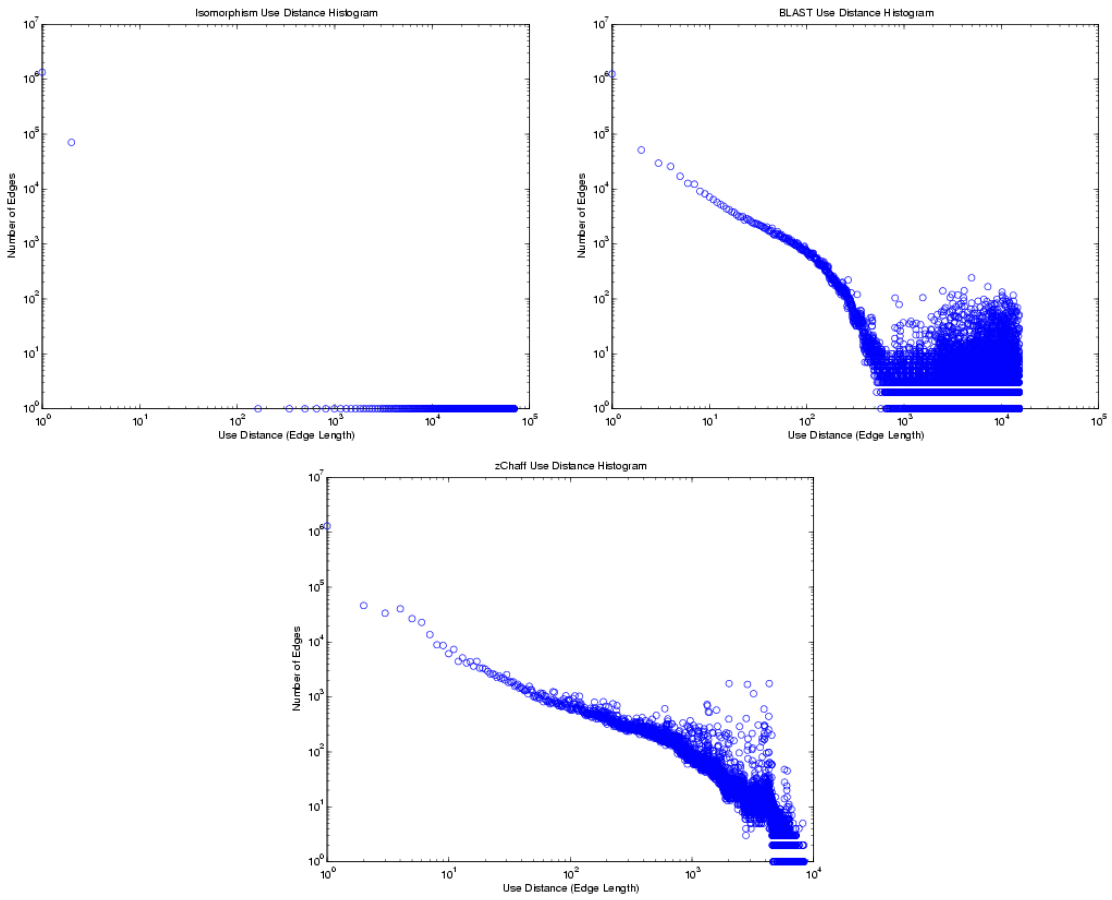


Figure B.8 (continued) Sandia Integer Suite Use Distance Histogram.

APPENDIX C

FULL THREADS RESULTS

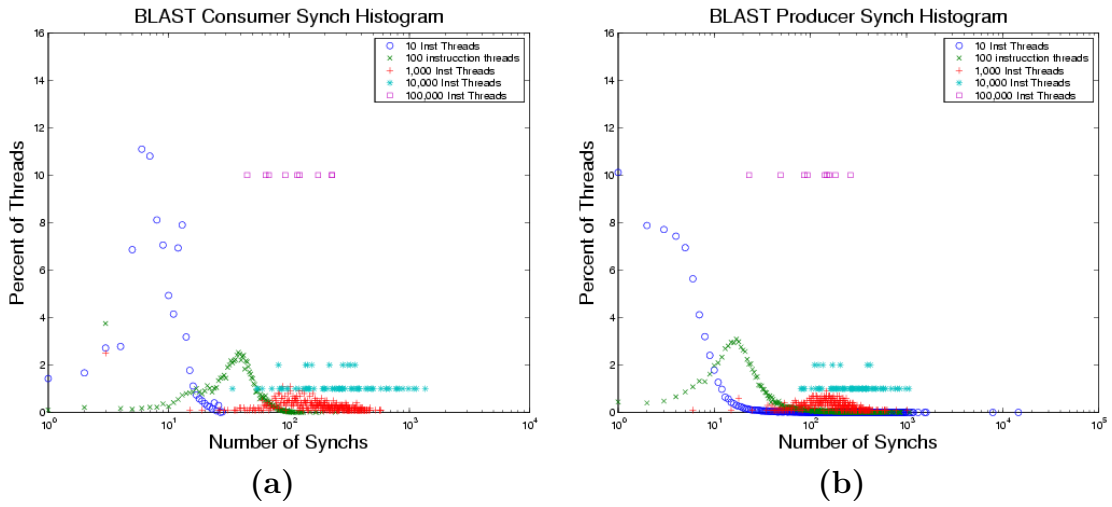


Figure C.1. BLAST Thread Properties (Continued on the next page.)

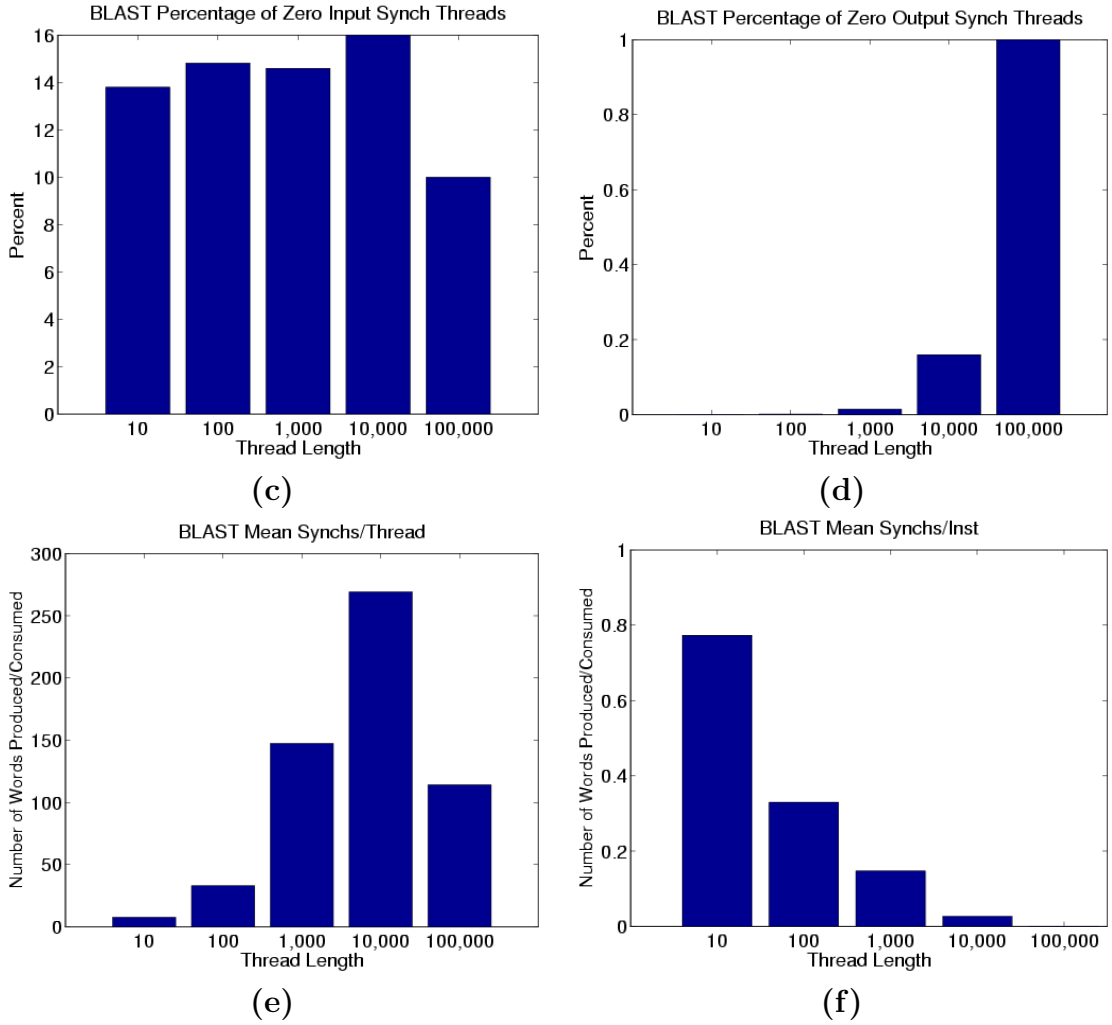


Figure C.1 (continued) BLAST Thread Properties (Continued on the next page.)

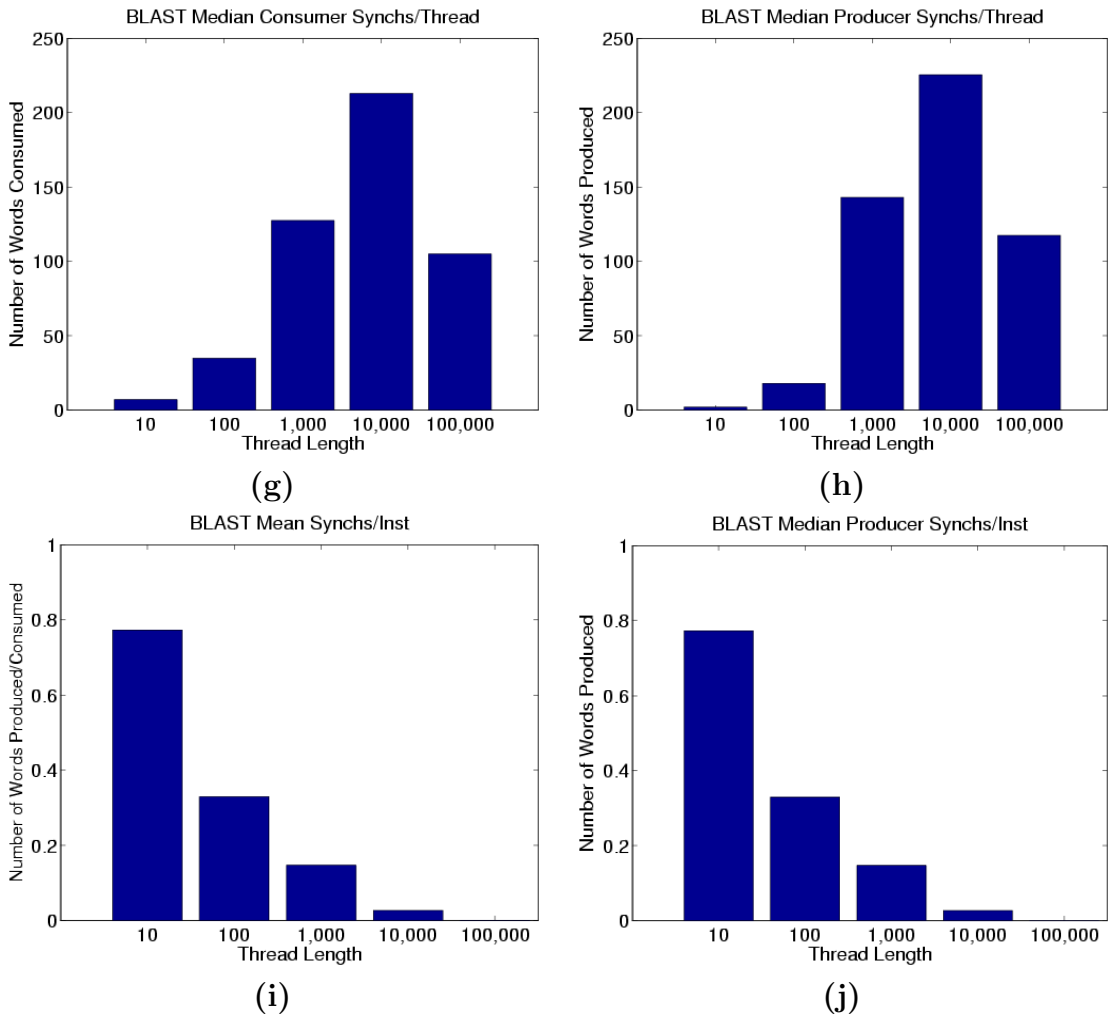
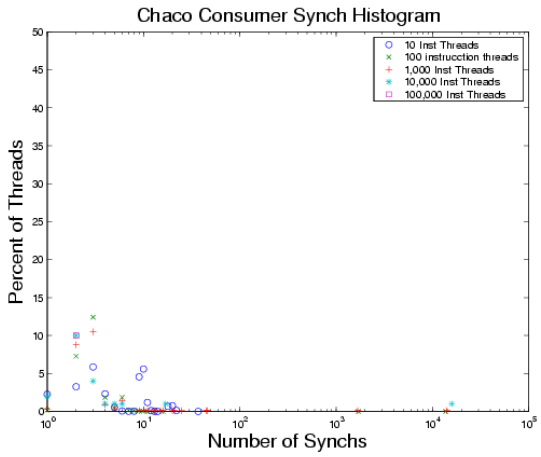
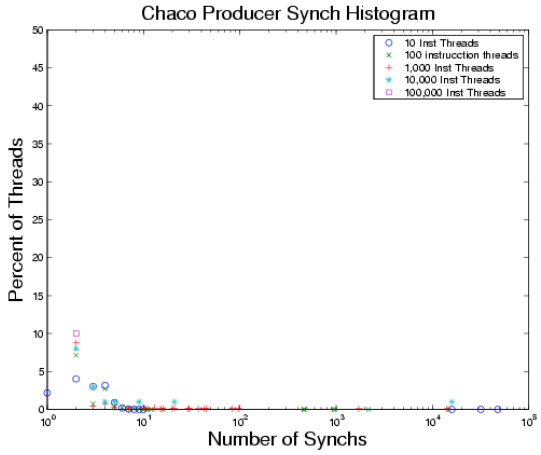


Figure C.1 (continued) BLAST Thread Properties



(a)



(b)

Figure C.2. Chaco Thread Properties (Continued on the next page.)

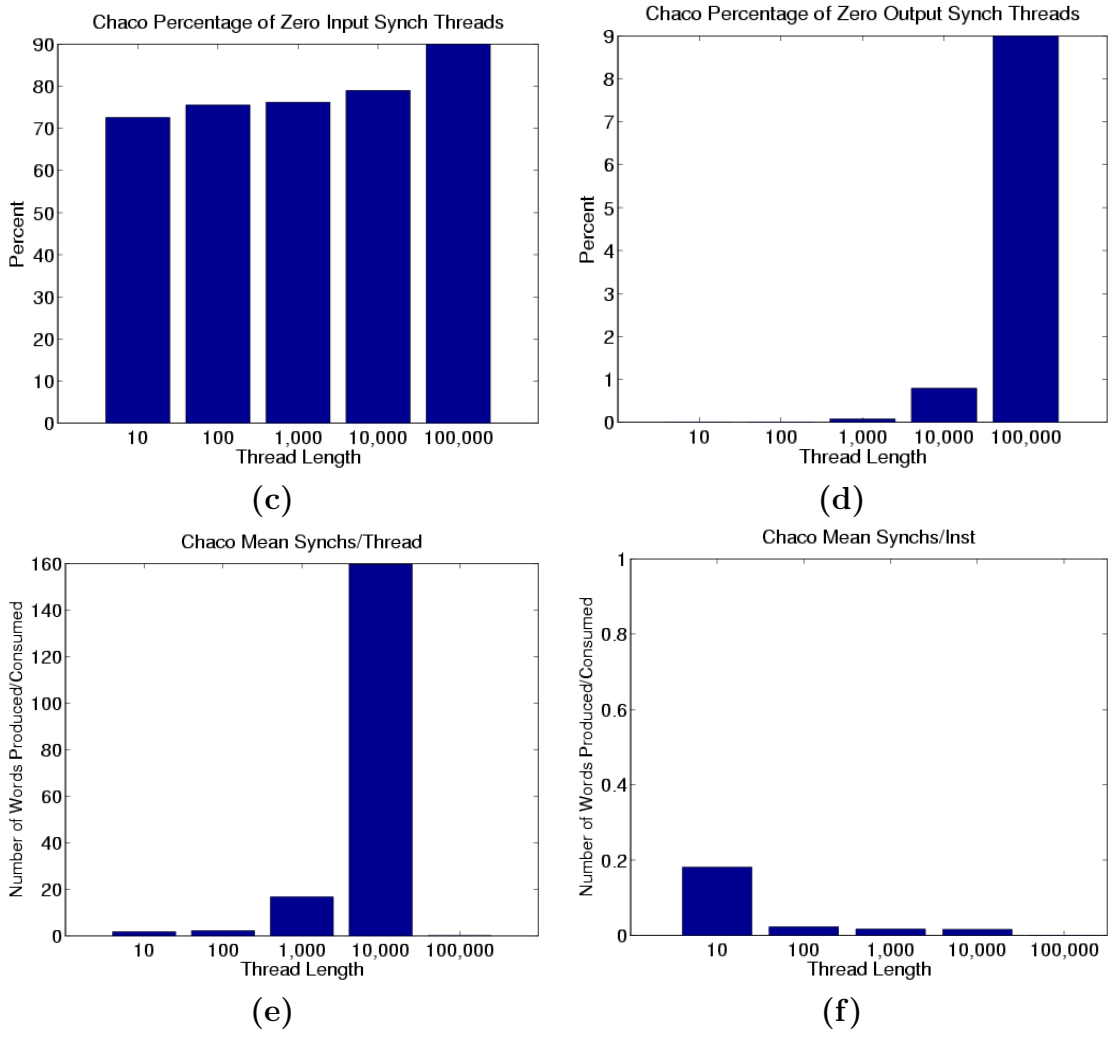


Figure C.2 (continued) Chaco Thread Properties (Continued on the next page.)

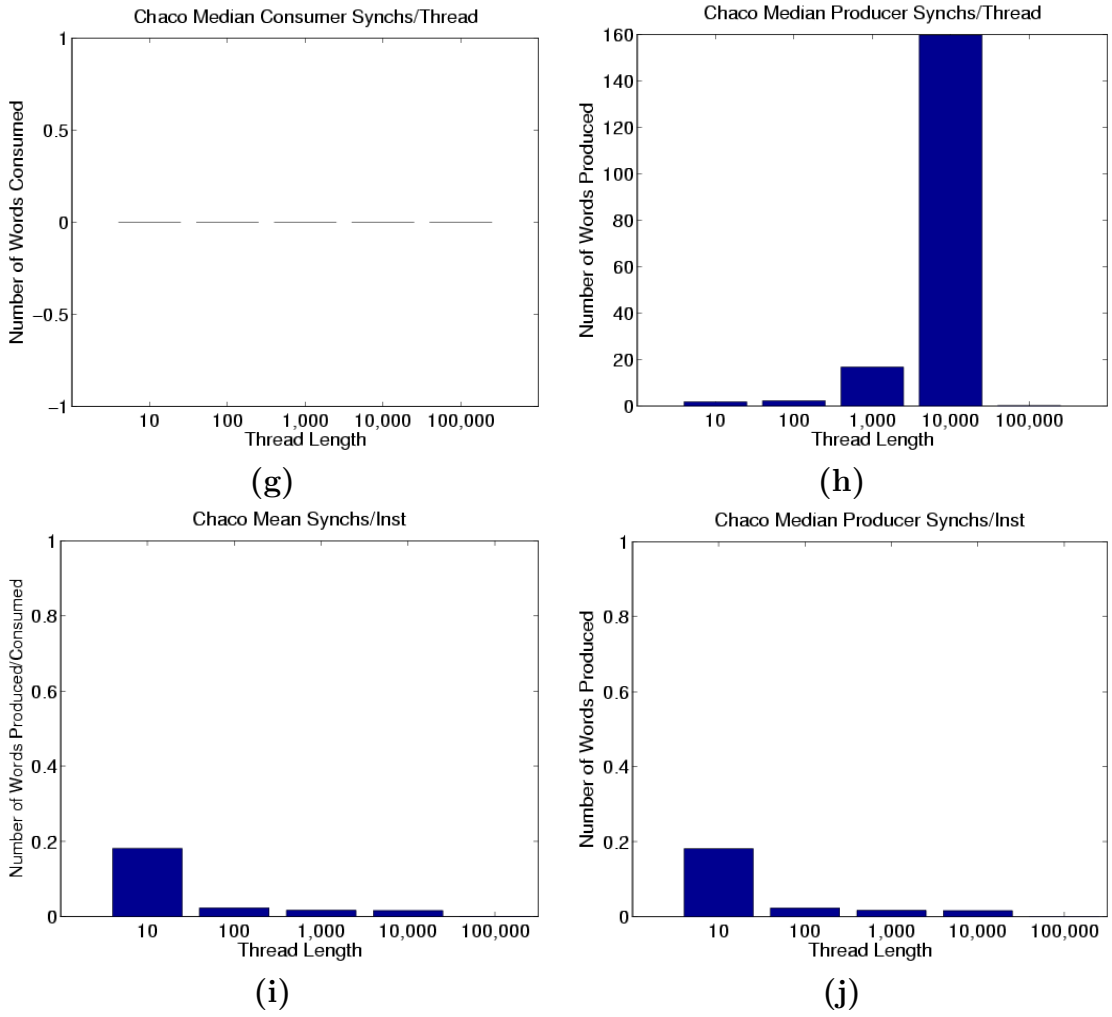
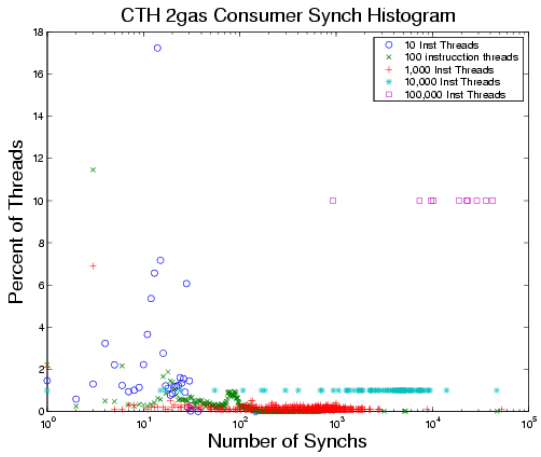
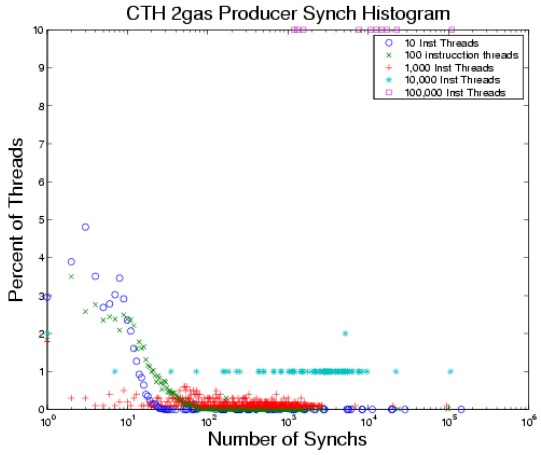


Figure C.2 (continued) Chaco Thread Properties



(a)



(b)

Figure C.3. CTH 2gas Thread Properties (Continued on the next page.)

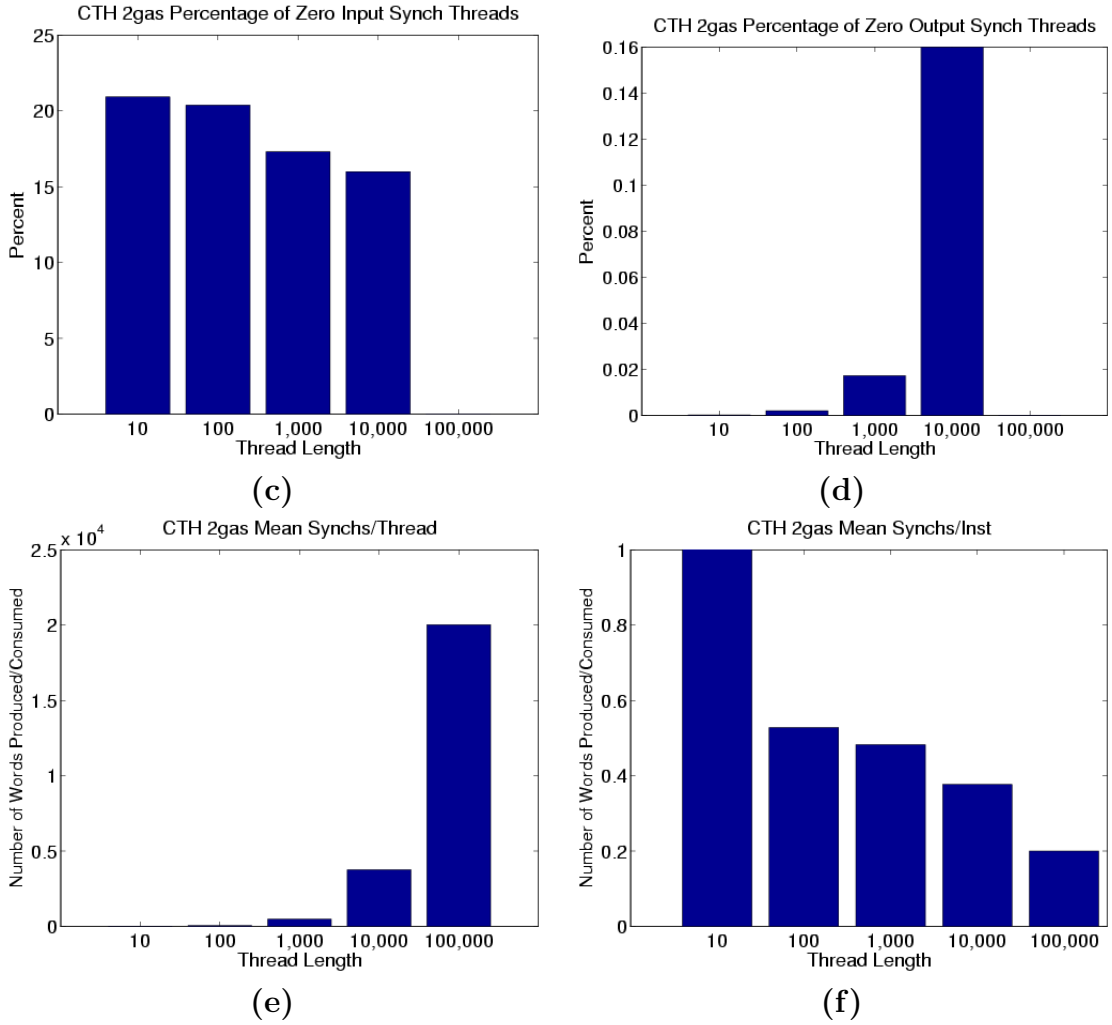


Figure C.3 (continued) CTH 2gas Thread Properties (Continued on the next page.)

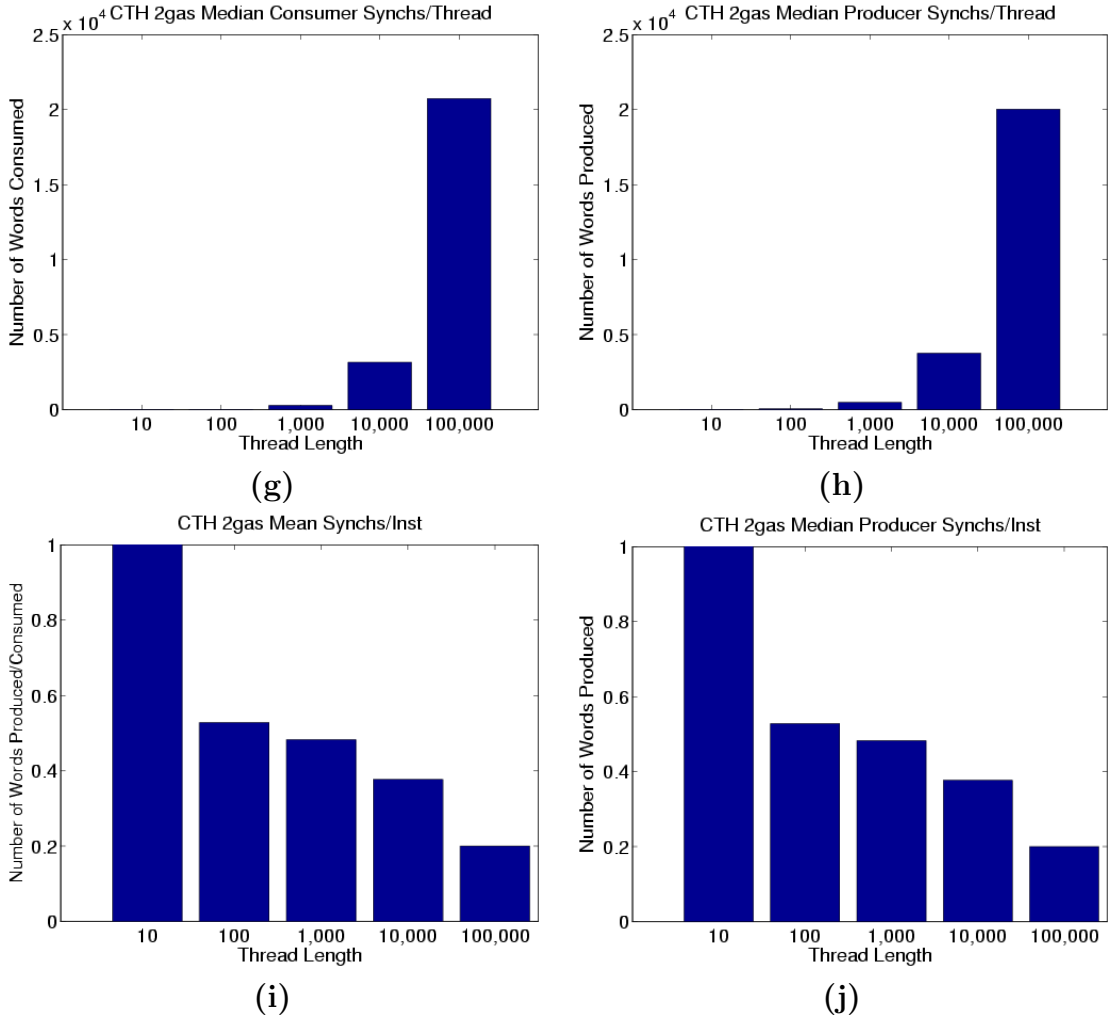
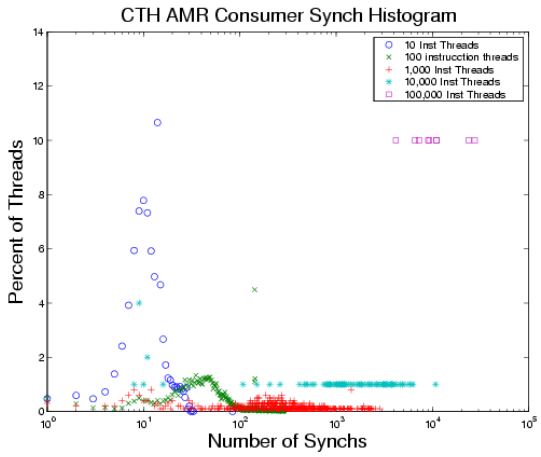
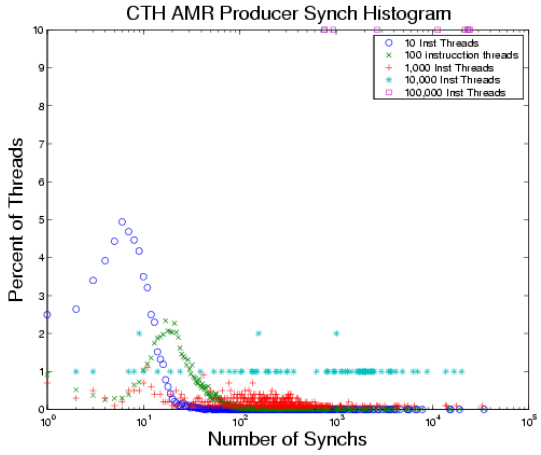


Figure C.3 (continued) CTH 2gas Thread Properties



(a)



(b)

Figure C.4. CTH AMR Thread Properties (Continued on the next page.)

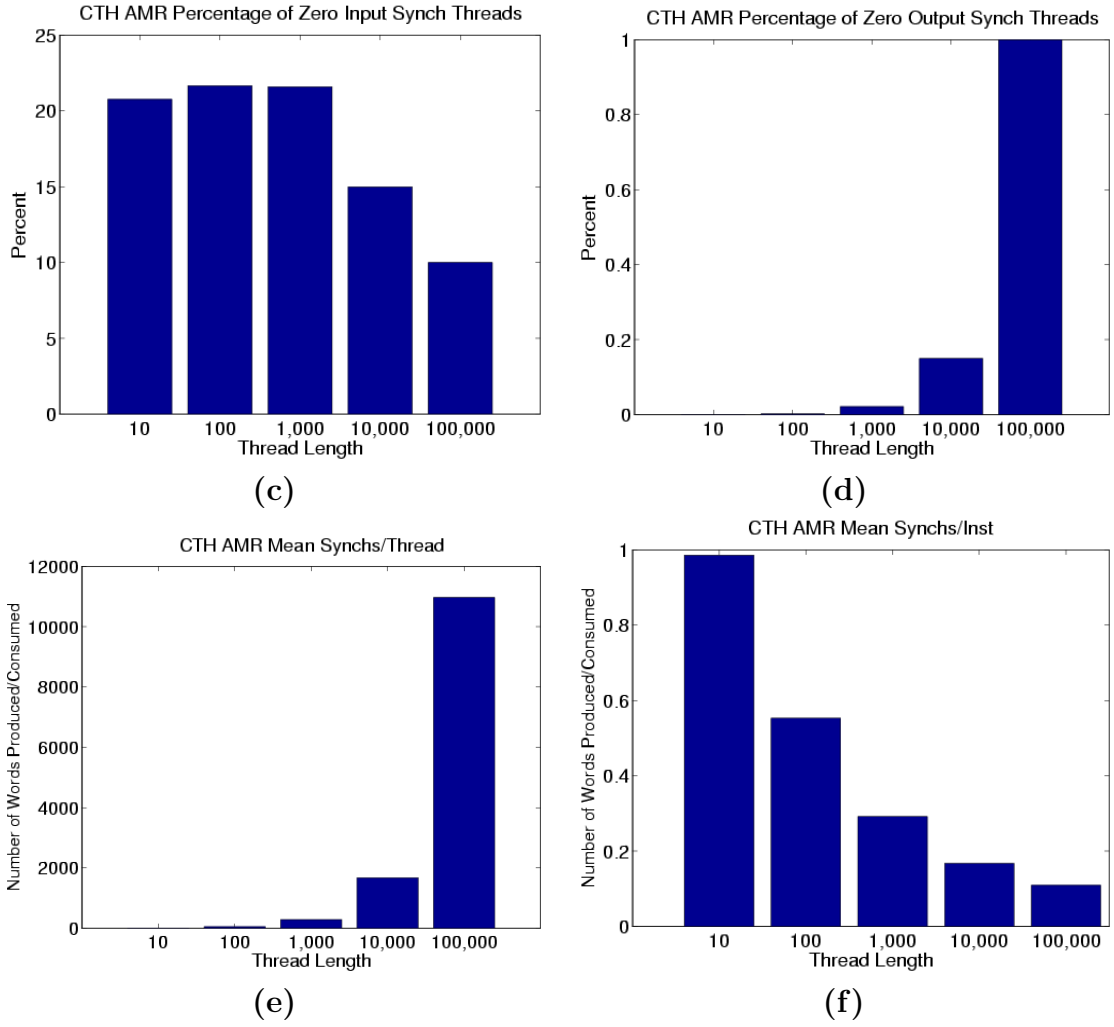


Figure C.4 (continued) CTH AMR Thread Properties (Continued on the next page.)

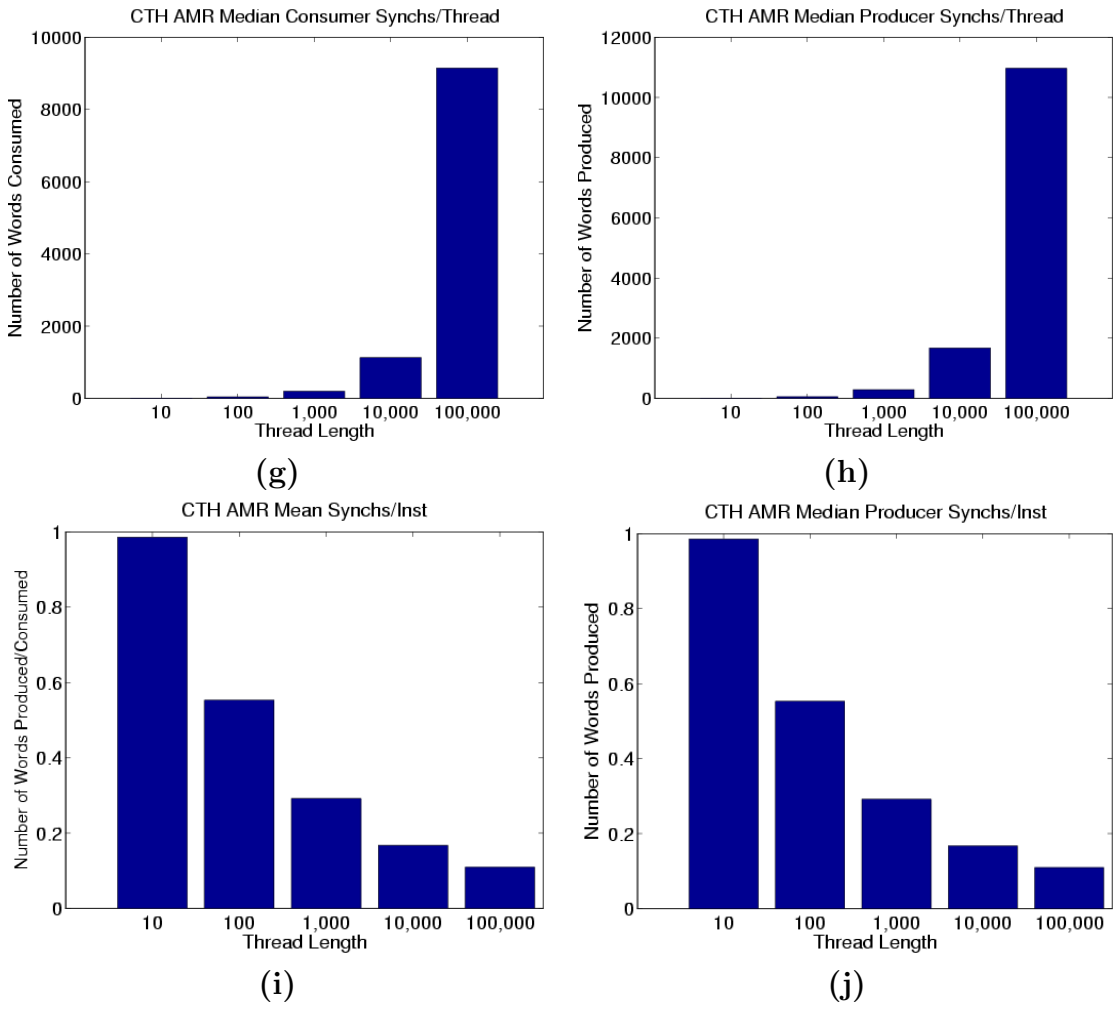
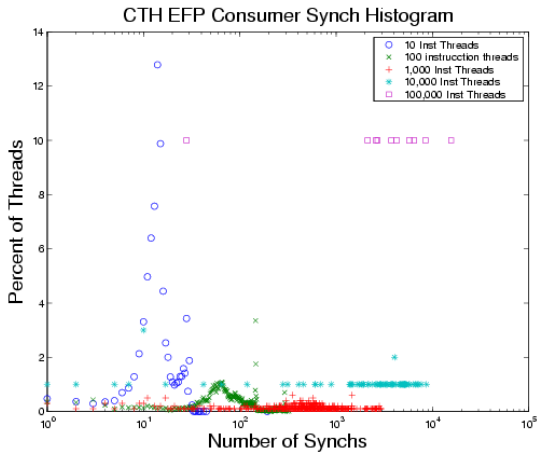
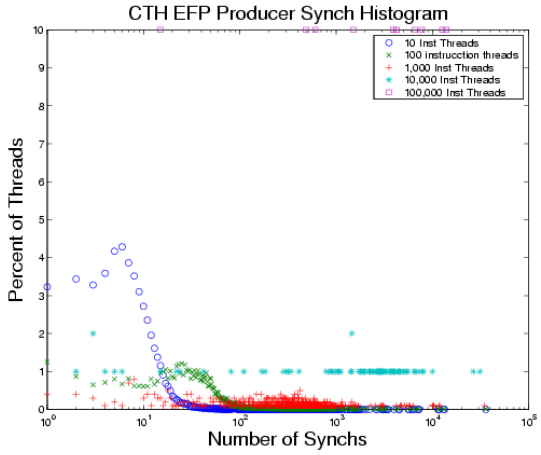


Figure C.4 (continued) CTH AMR Thread Properties



(a)



(b)

Figure C.5. CTH EFP Thread Properties (Continued on the next page.)

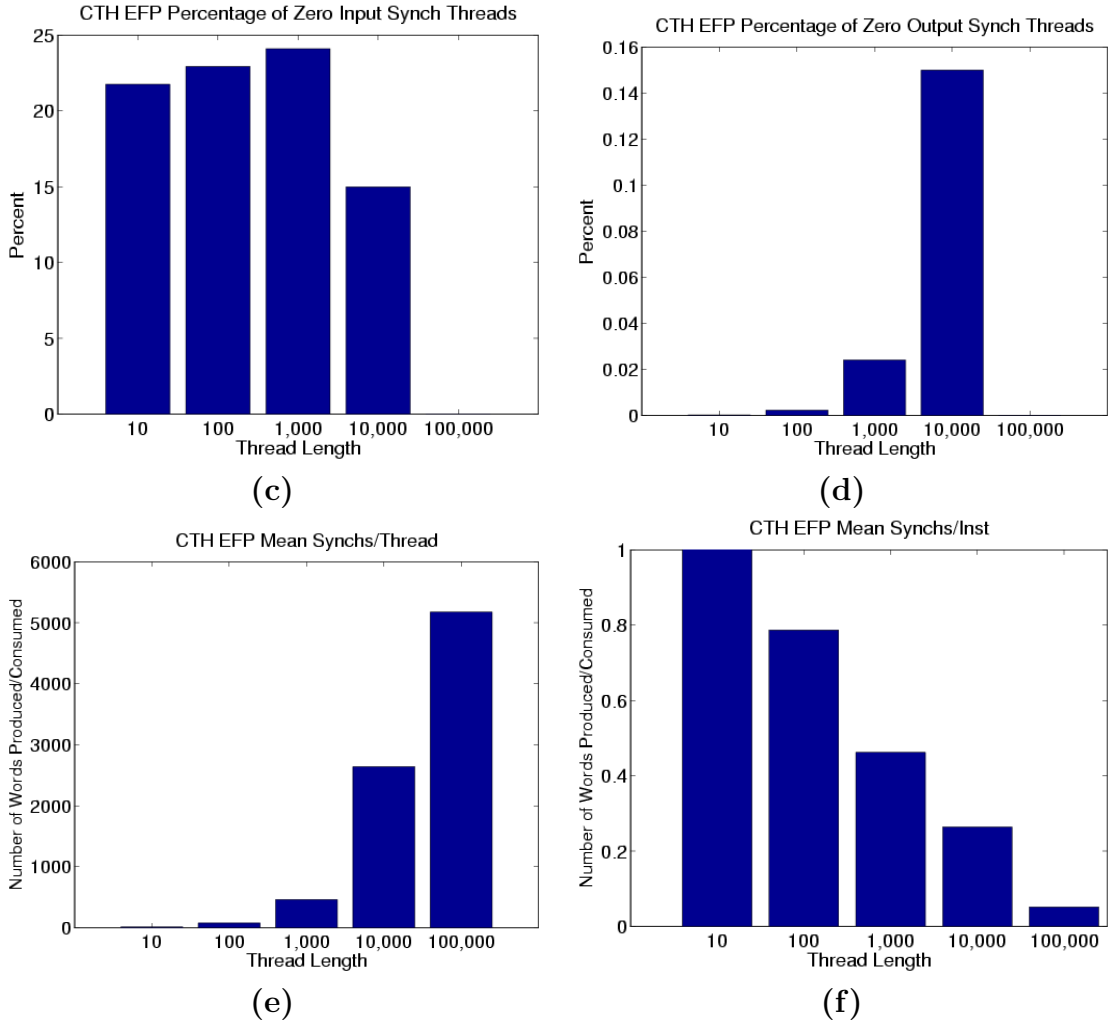


Figure C.5 (continued) CTH EFP Thread Properties (Continued on the next page.)

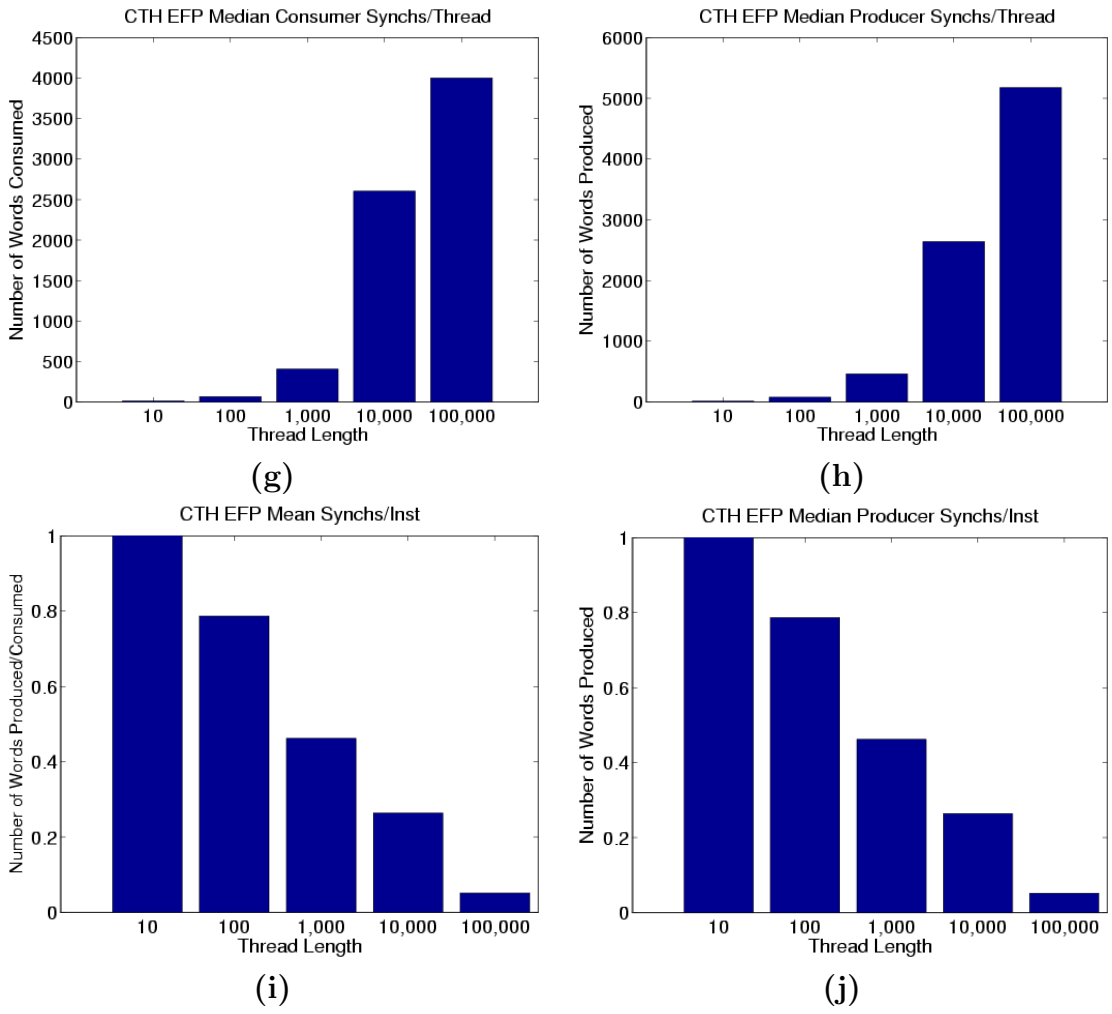
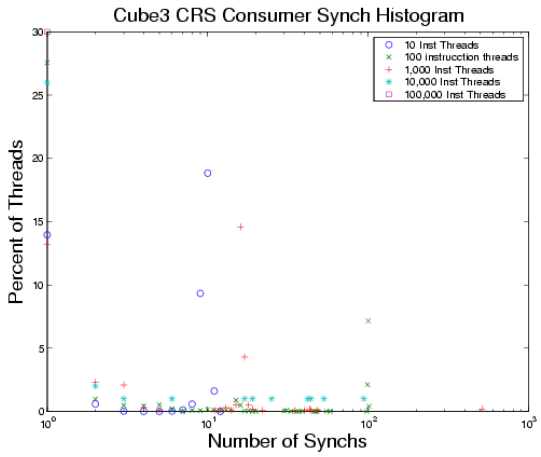
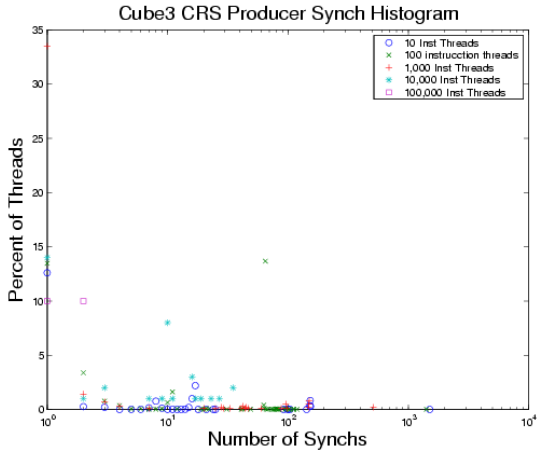


Figure C.5 (continued) CTH EFP Thread Properties



(a)



(b)

Figure C.6. Cube3 CRS Thread Properties (Continued on the next page.)

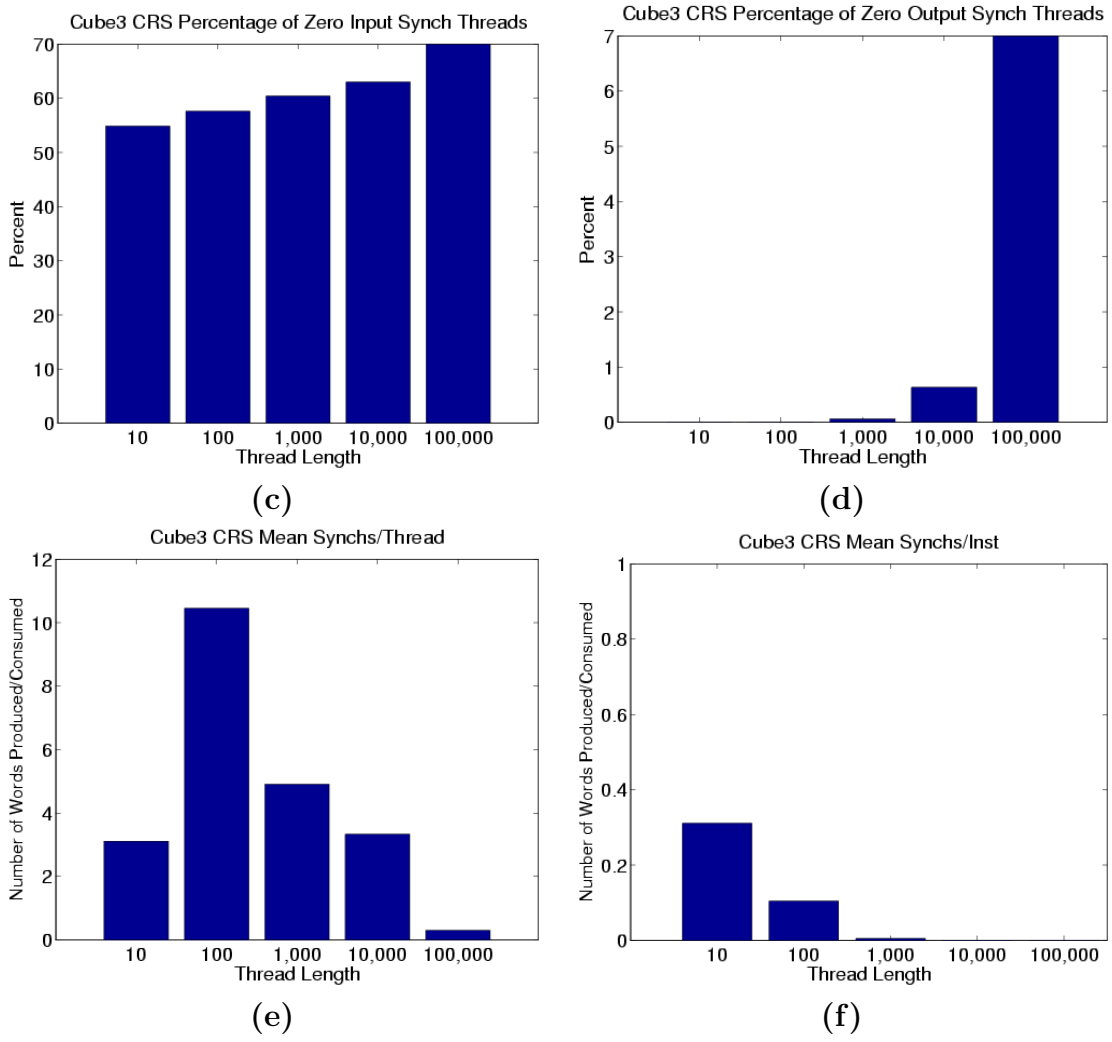


Figure C.6 (continued) Cube3 CRS Thread Properties (Continued on the next page.)

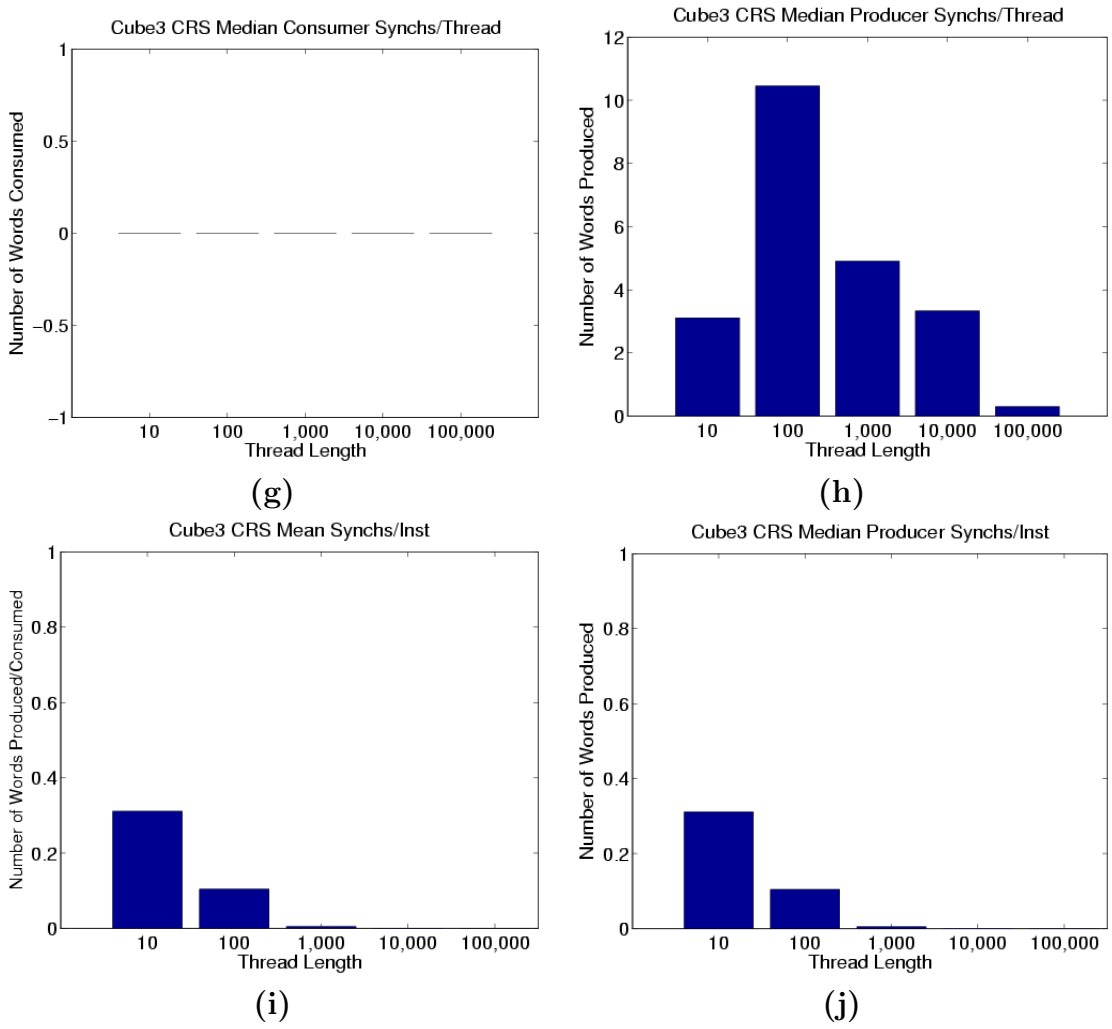
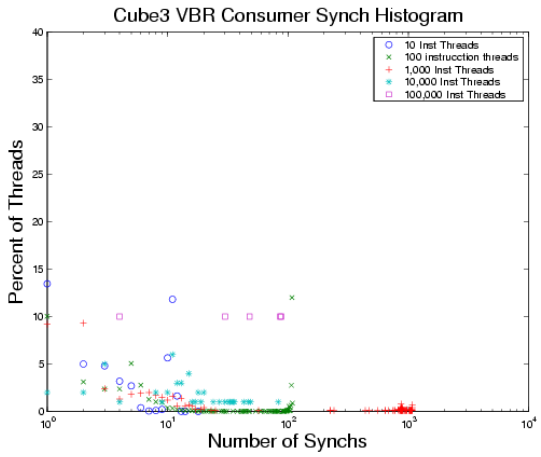
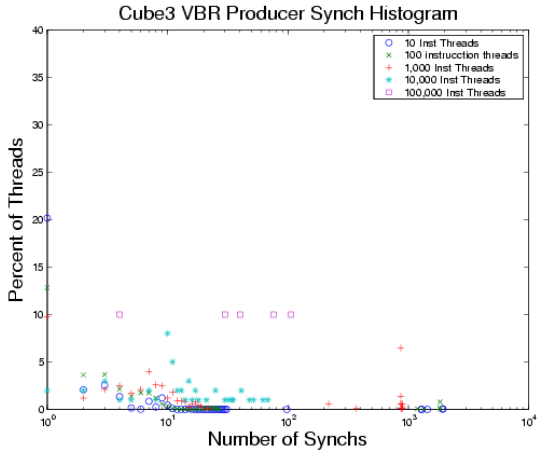


Figure C.6 (continued) Cube3 CRS Thread Properties



(a)



(b)

Figure C.7. Cube3 VBR Thread Properties (Continued on the next page.)

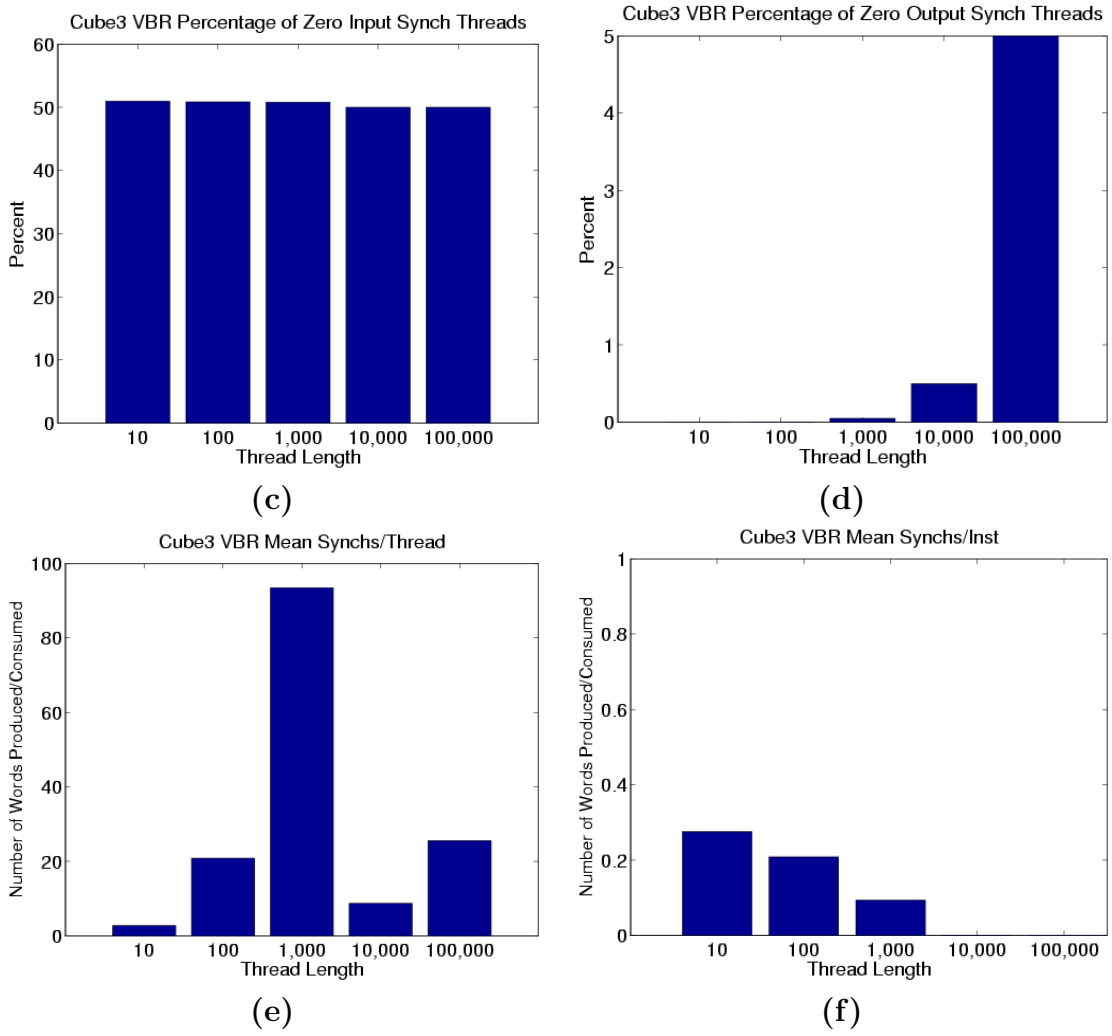


Figure C.7 (continued) Cube3 VBR Thread Properties (Continued on the next page.)

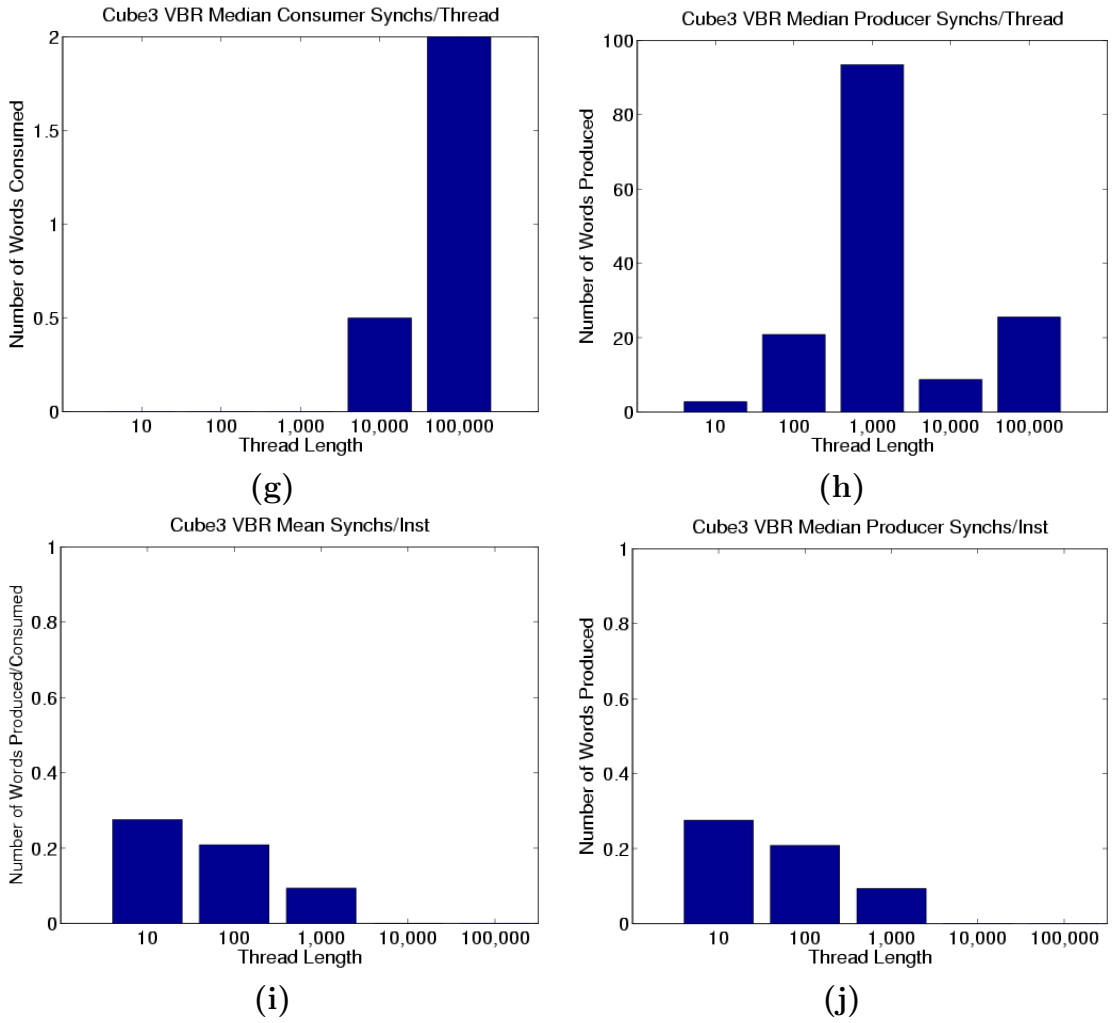
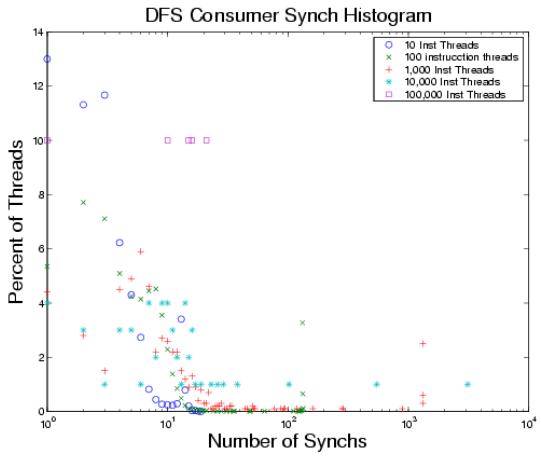
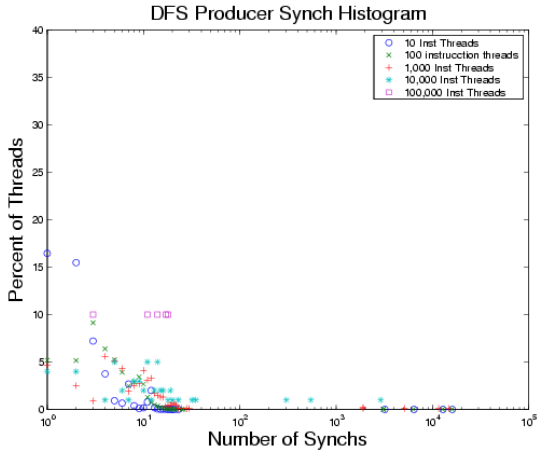


Figure C.7 (continued) Cube3 VBR Thread Properties



(a)



(b)

Figure C.8. DFS Thread Properties (Continued on the next page.)

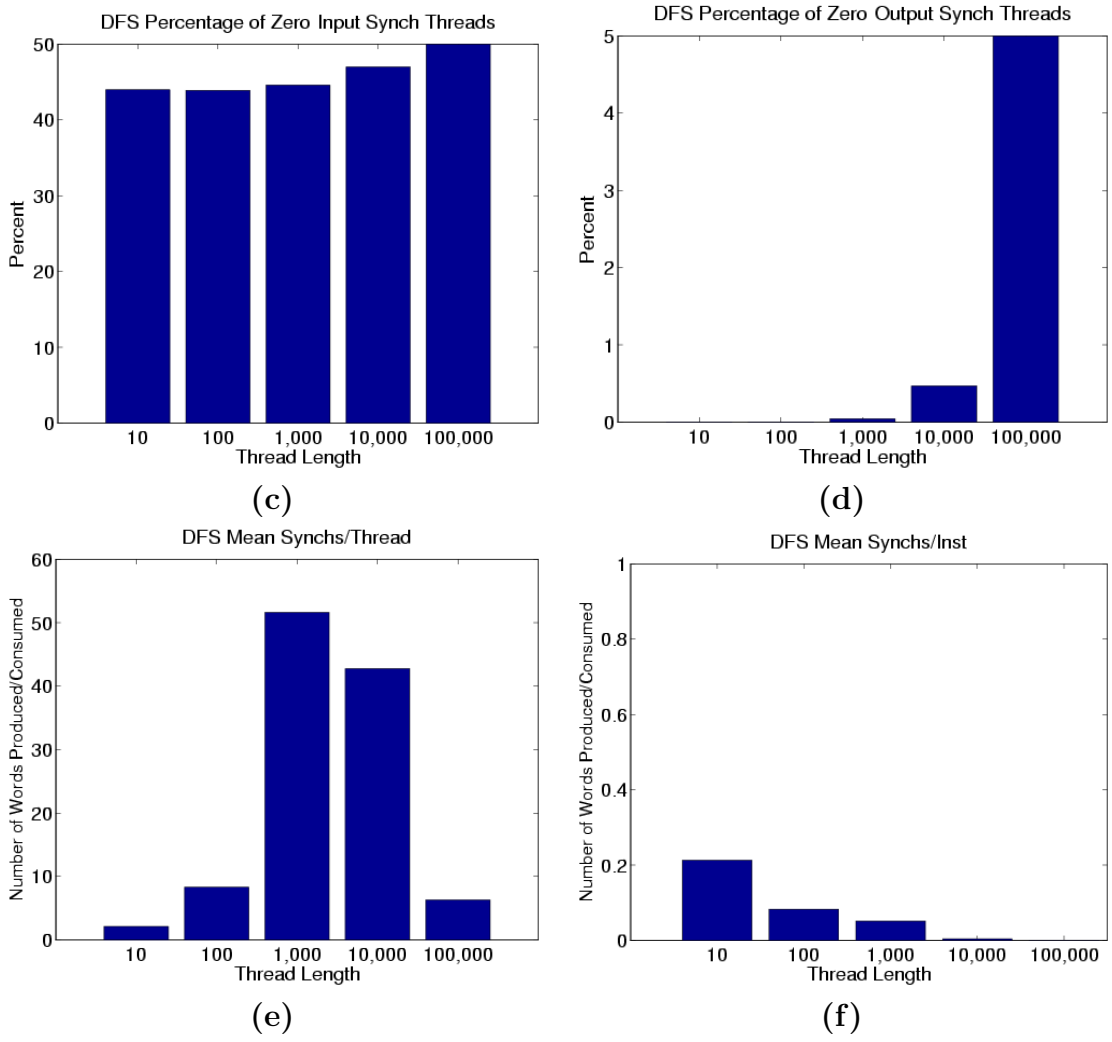


Figure C.8 (continued) DFS Thread Properties (Continued on the next page.)

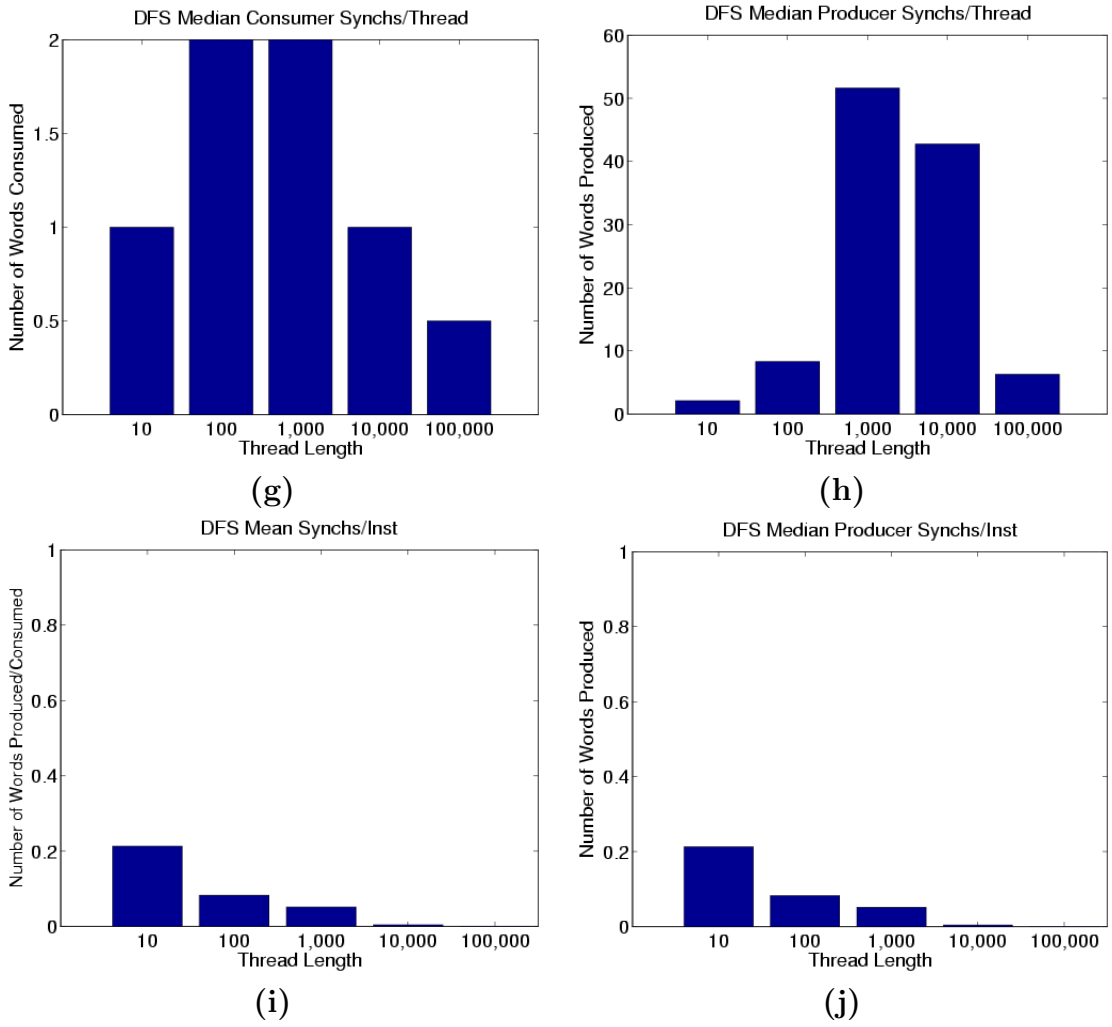
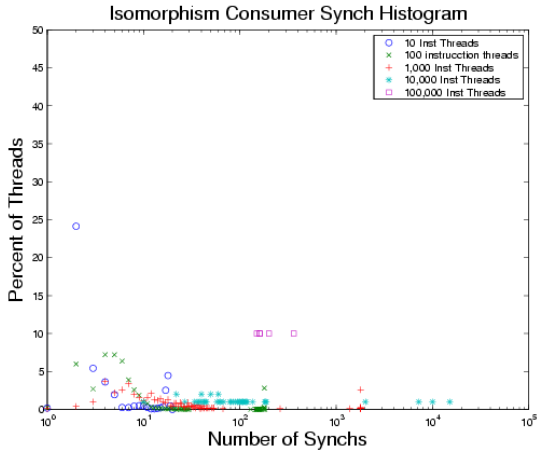
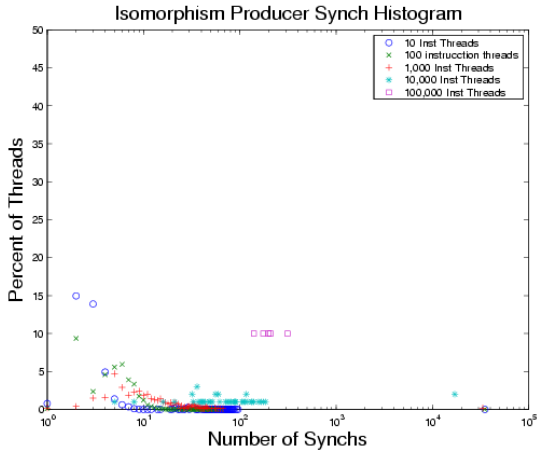


Figure C.8 (continued) DFS Thread Properties



(a)



(b)

Figure C.9. Isomorphism Thread Properties (Continued on the next page.)

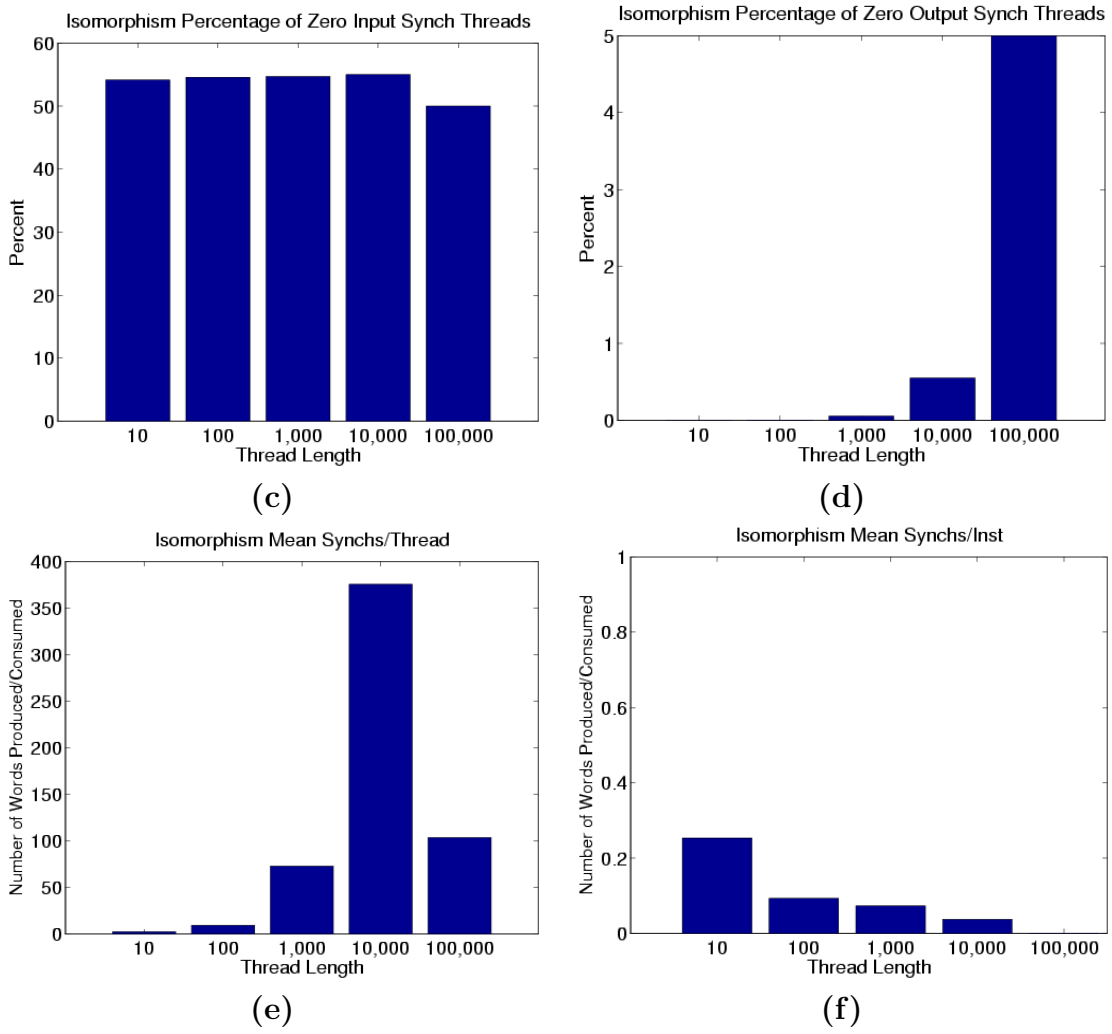


Figure C.9 (continued) Isomorphism Thread Properties (Continued on the next page.)

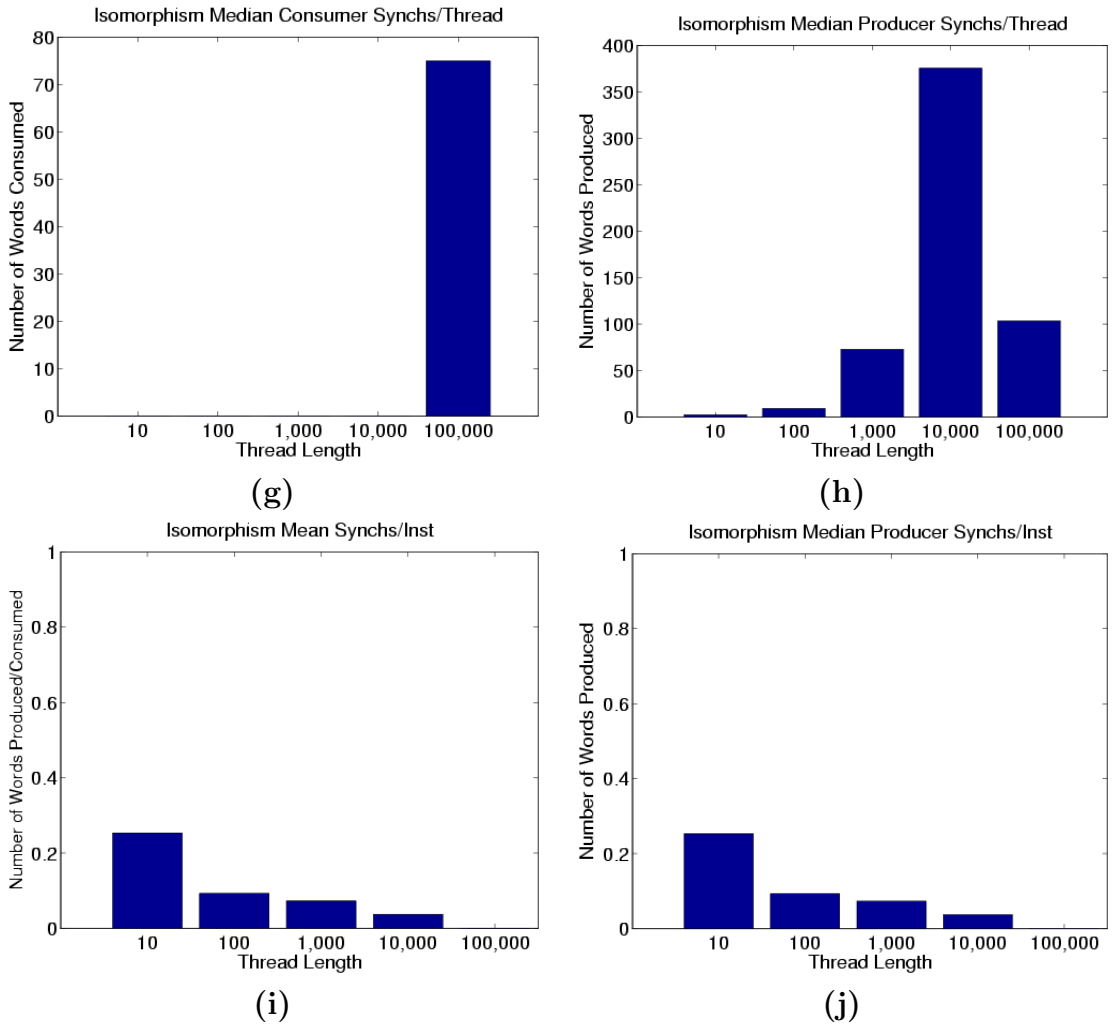
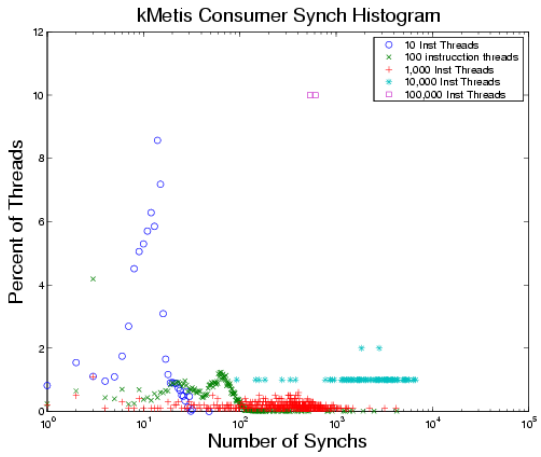
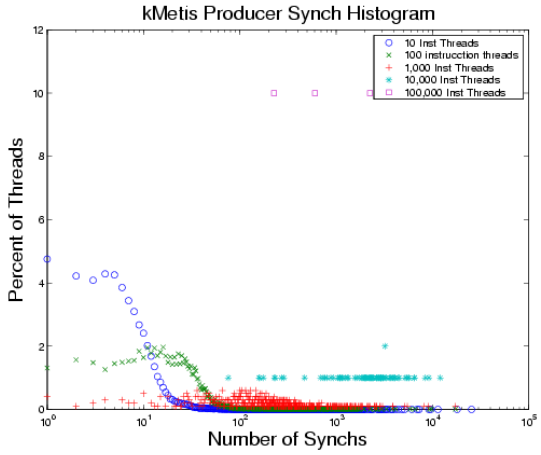


Figure C.9 (continued) Isomorphism Thread Properties



(a)



(b)

Figure C.10. kMetis Thread Properties (Continued on the next page.)

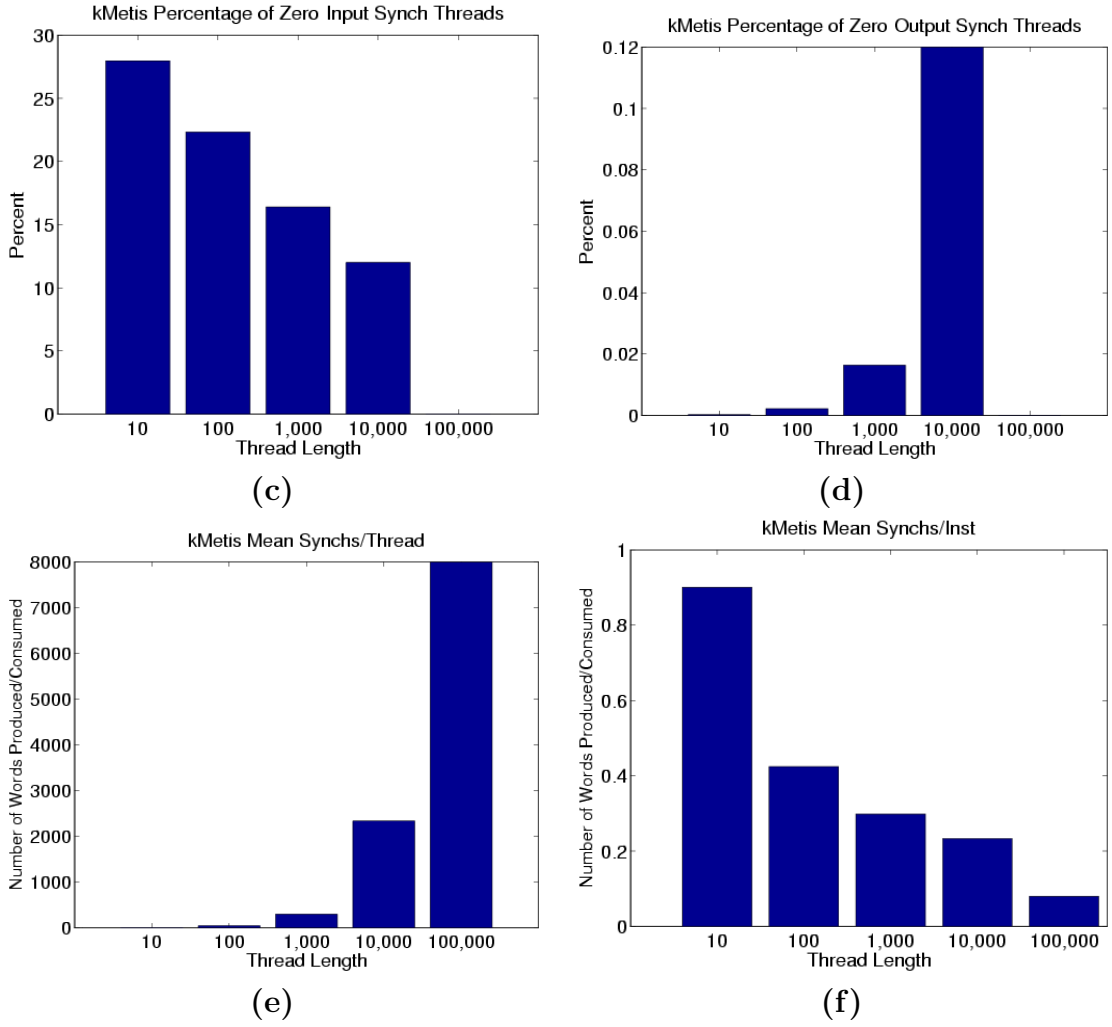


Figure C.10 (continued) kMetis Thread Properties (Continued on the next page.)

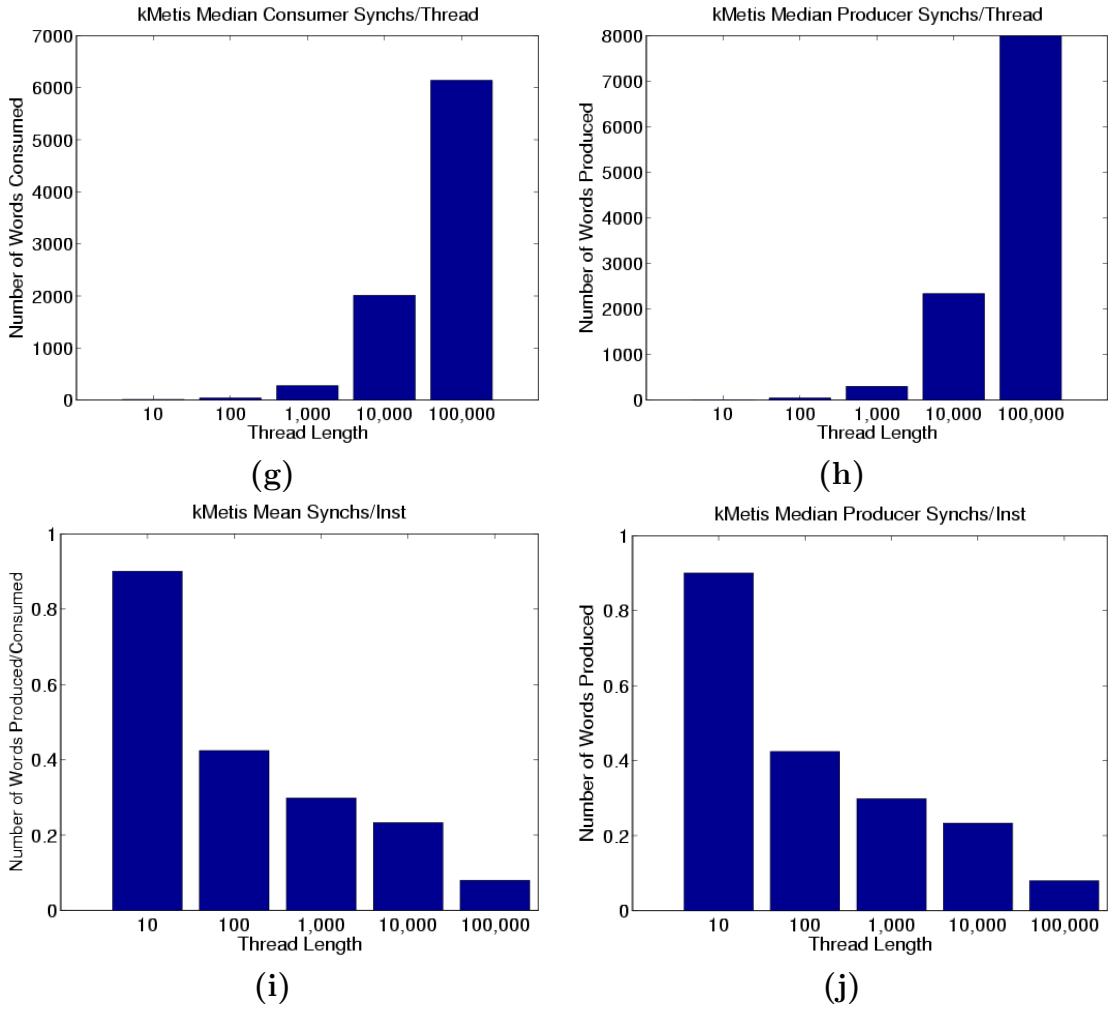
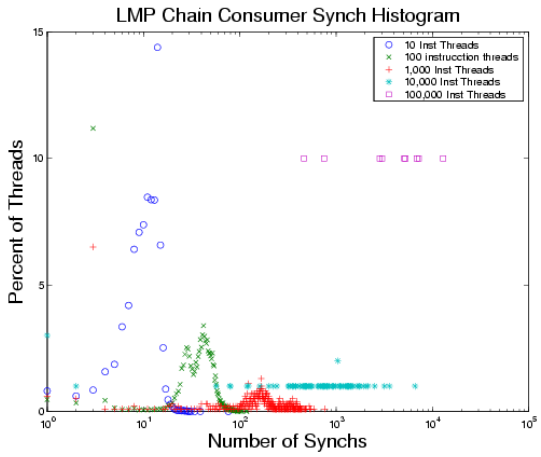
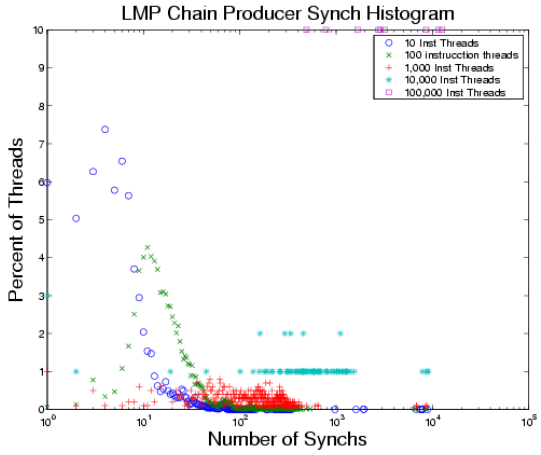


Figure C.10 (continued) kMetis Thread Properties



(a)



(b)

Figure C.11. LMP Chain Thread Properties (Continued on the next page.)

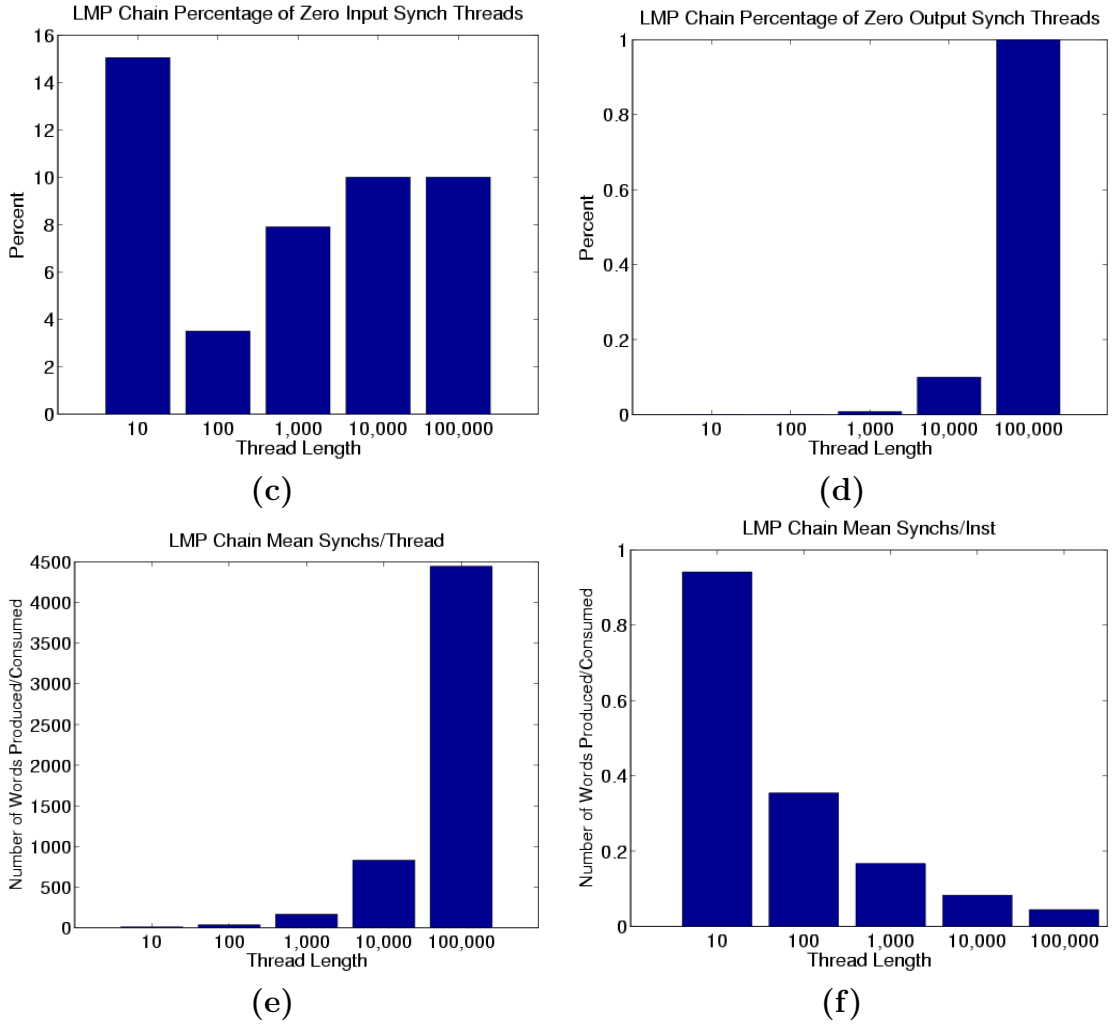


Figure C.11 (continued) LMP Chain Thread Properties (Continued on the next page.)

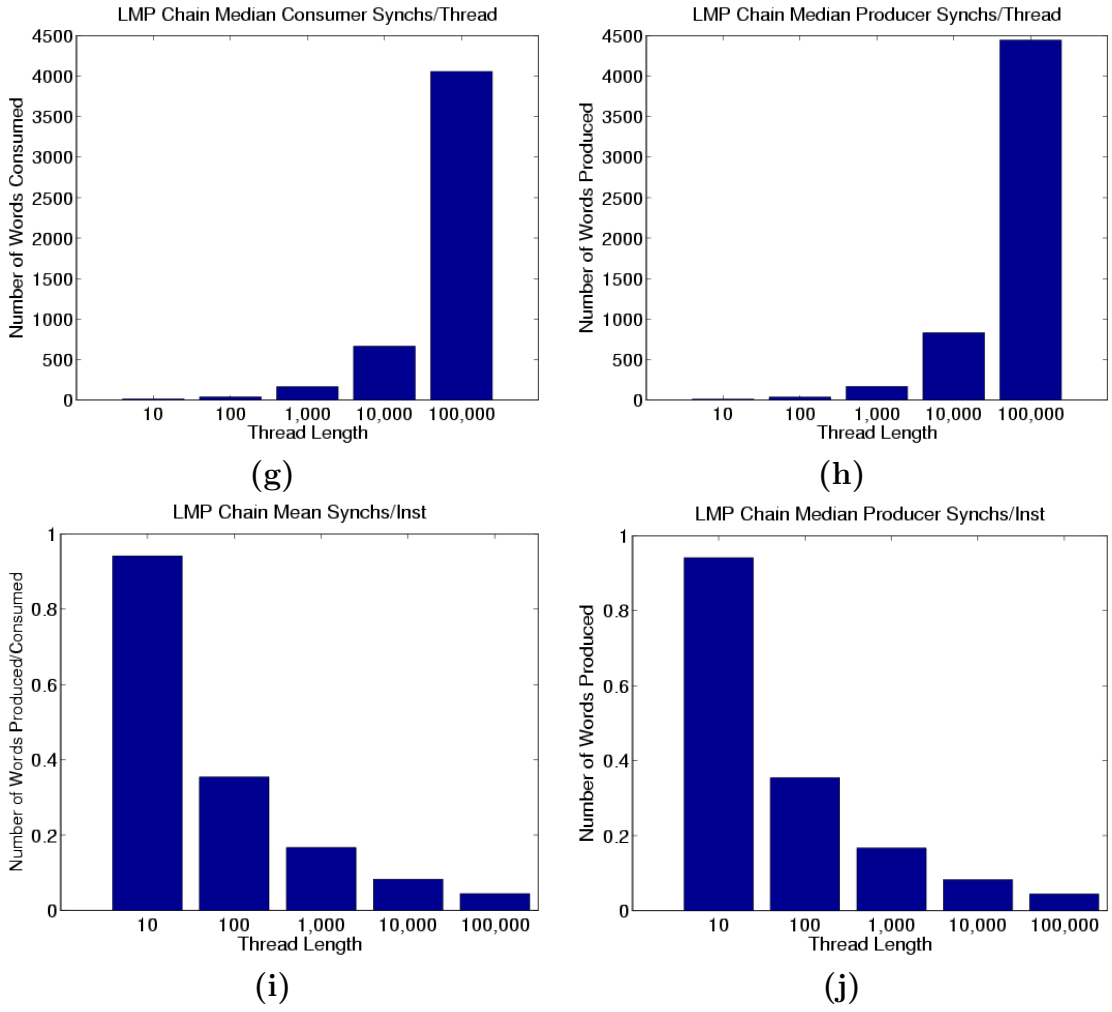
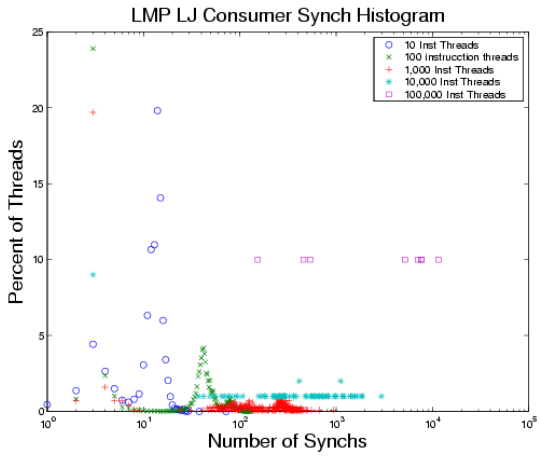
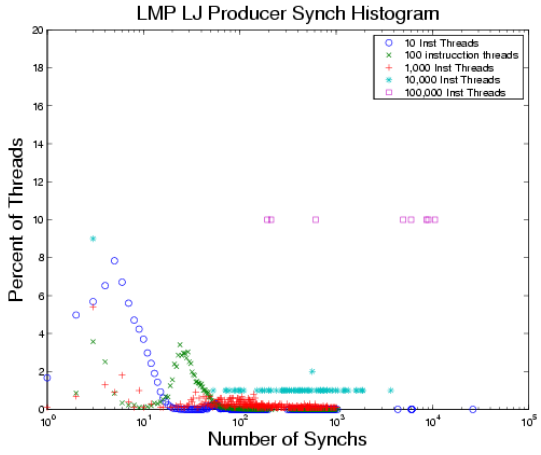


Figure C.11 (continued) LMP Chain Thread Properties



(a)



(b)

Figure C.12. LMP LJ Thread Properties (Continued on the next page.)

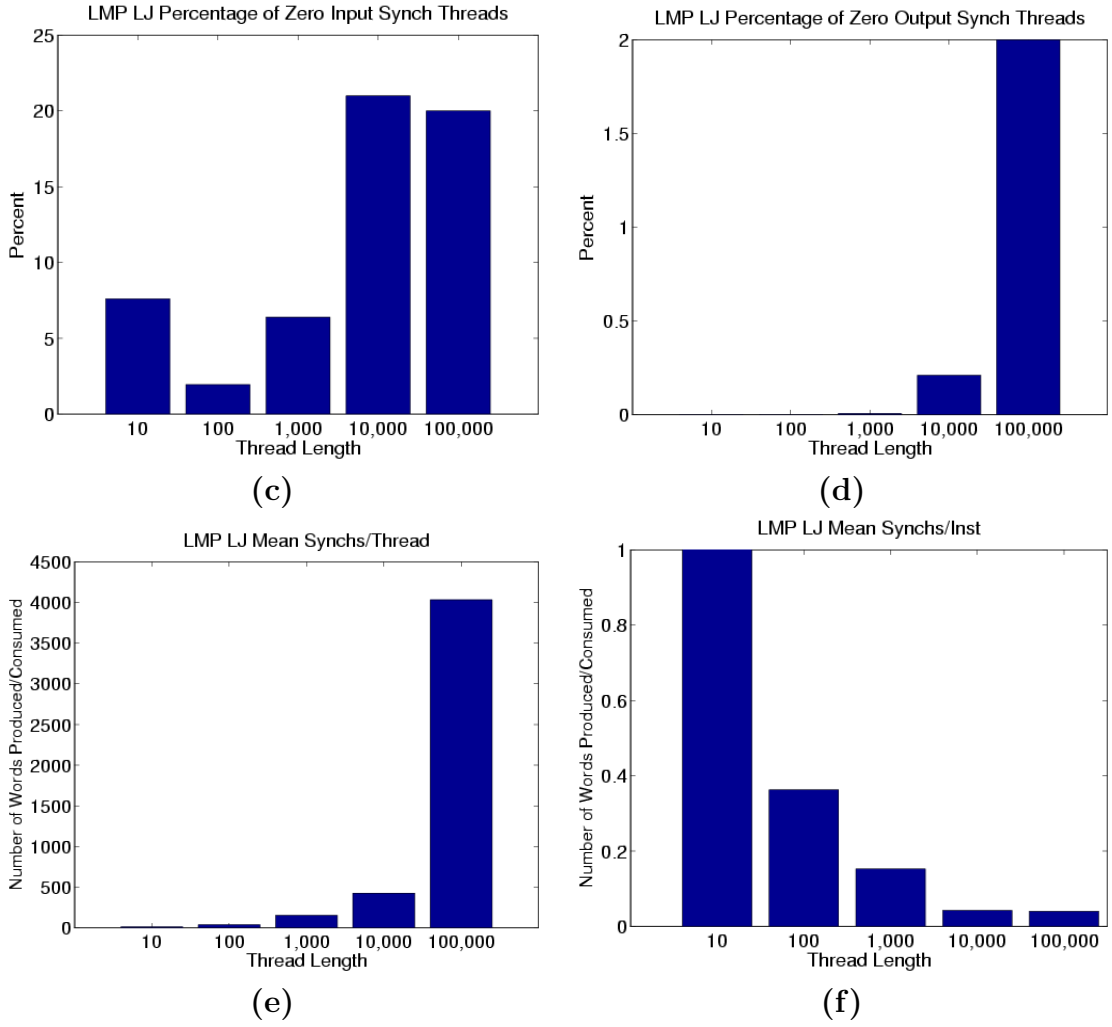


Figure C.12 (continued) LMP LJ Thread Properties (Continued on the next page.)

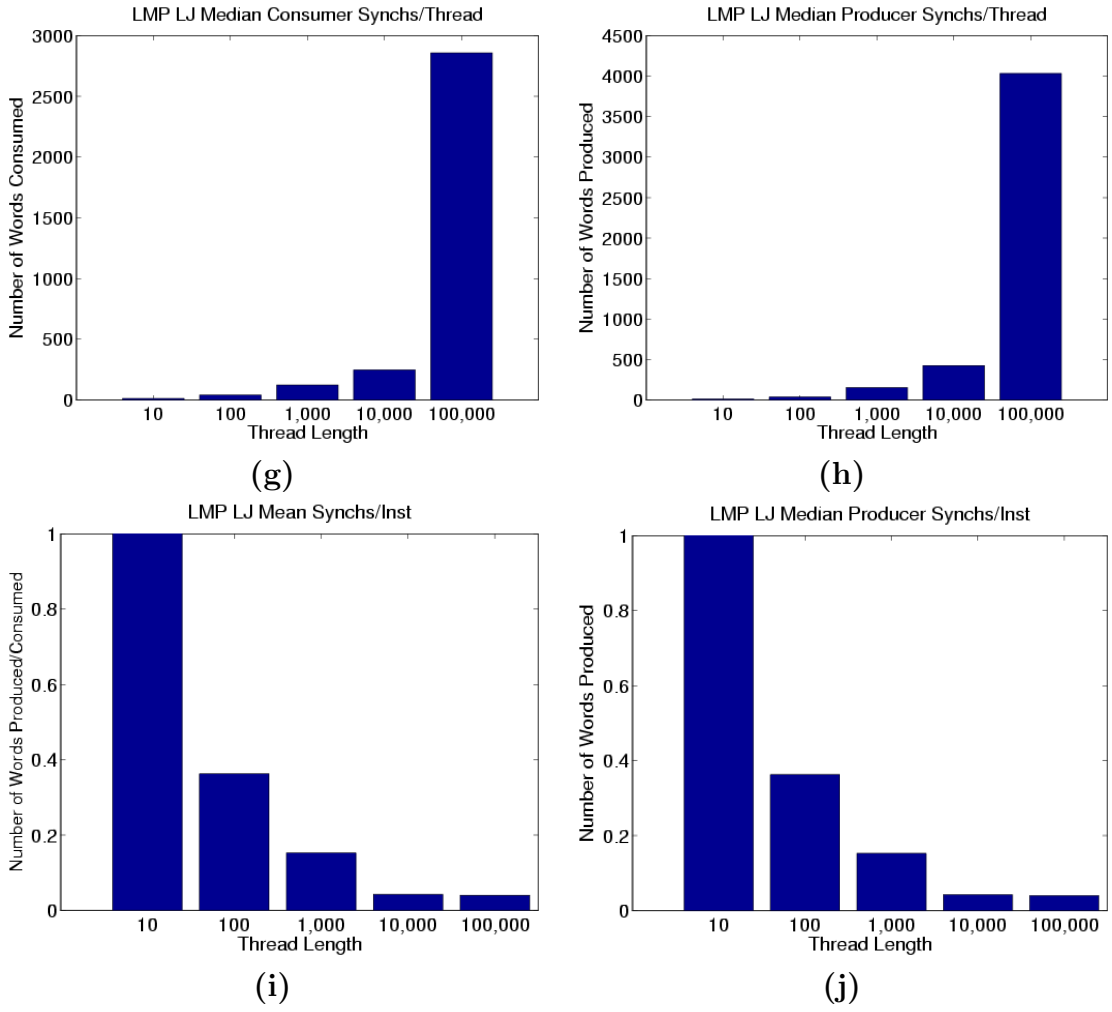
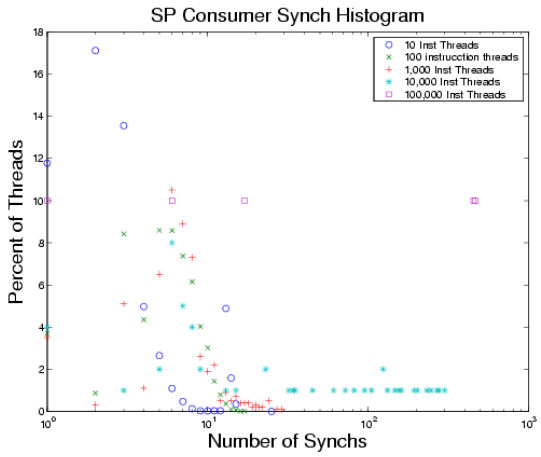
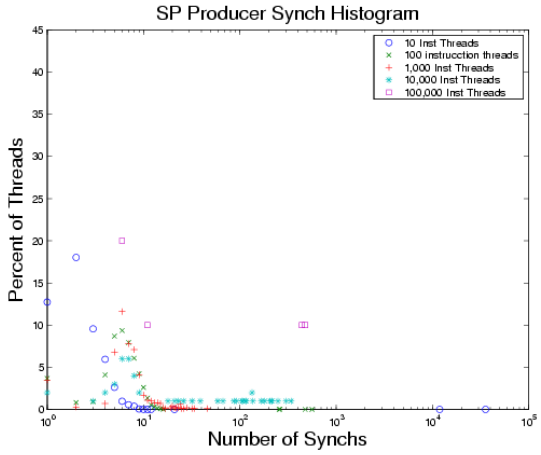


Figure C.12 (continued) LMP LJ Thread Properties



(a)



(b)

Figure C.13. SP Thread Properties (Continued on the next page.)

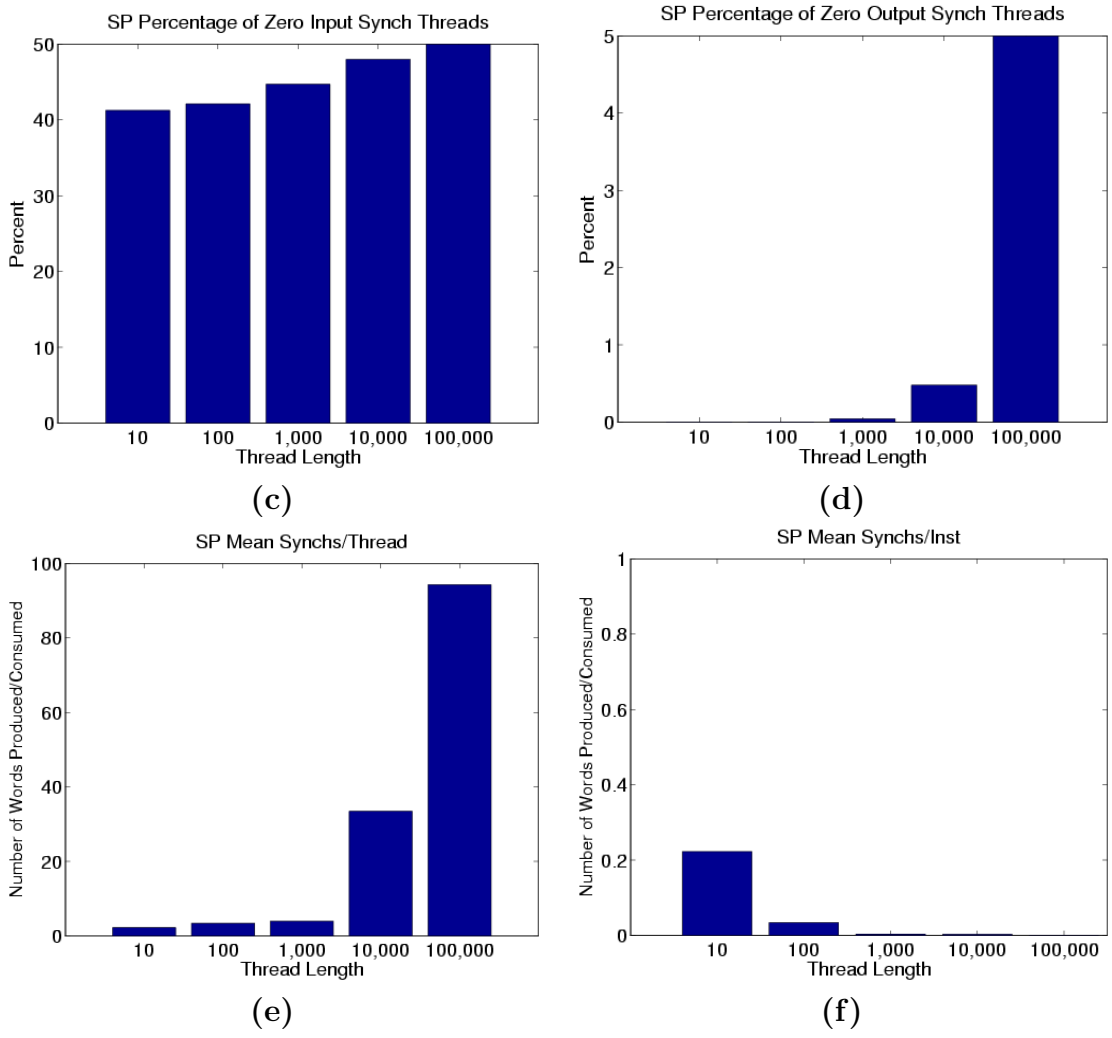


Figure C.13 (continued) SP Thread Properties (Continued on the next page.)

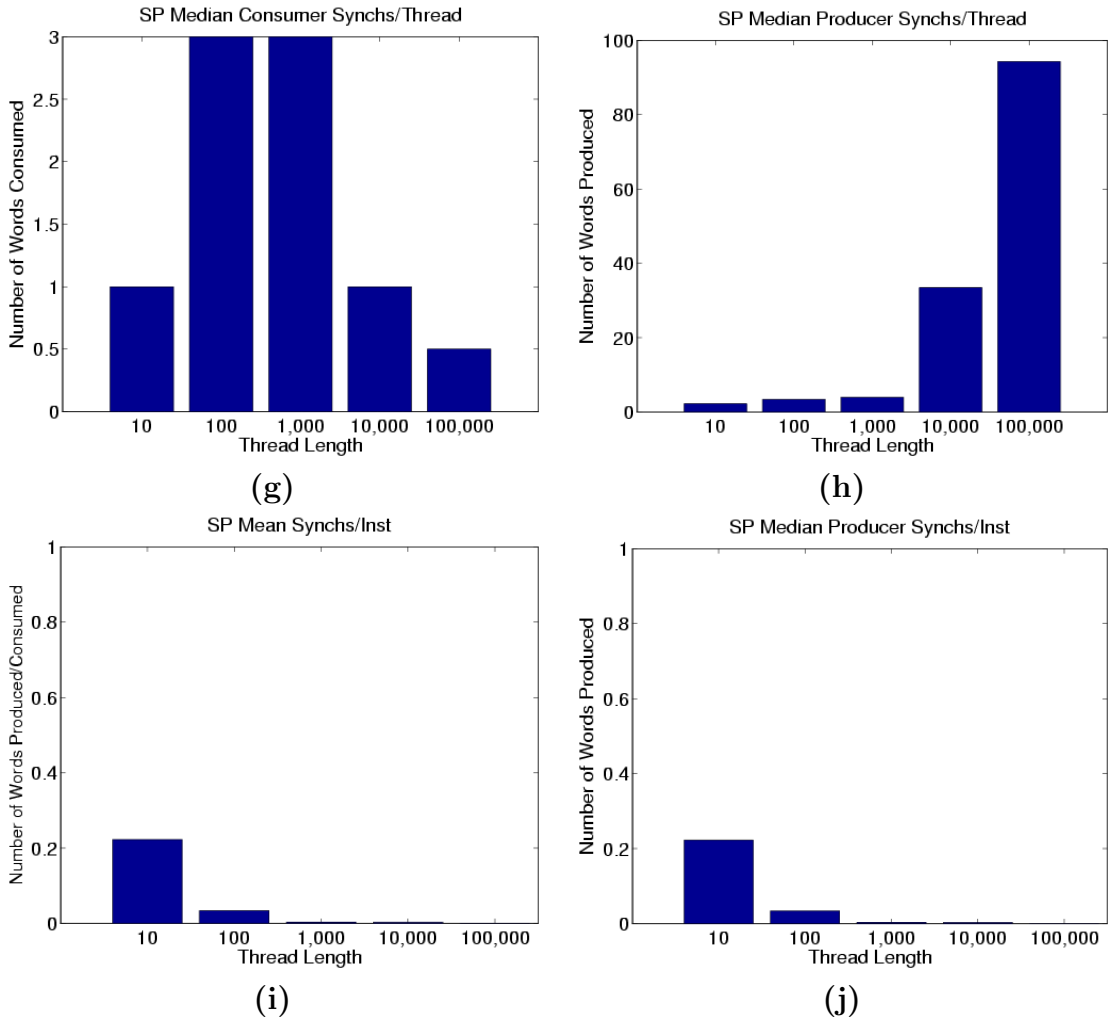
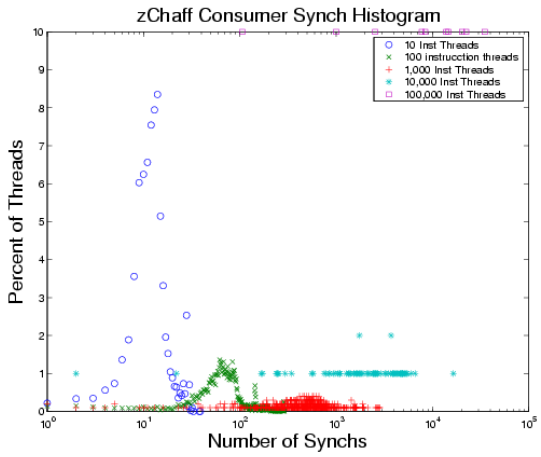
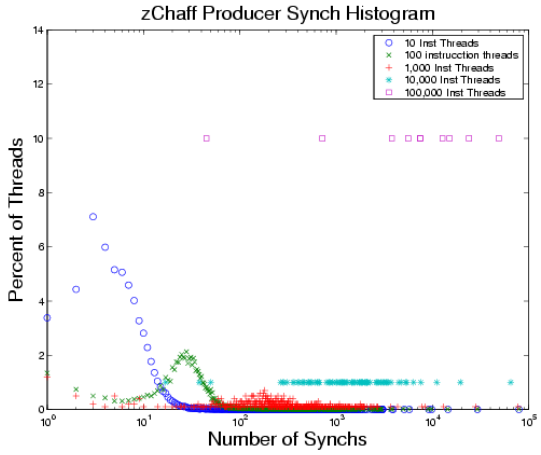


Figure C.13 (continued) SP Thread Properties



(a)



(b)

Figure C.14. zChaff Thread Properties (Continued on the next page.)

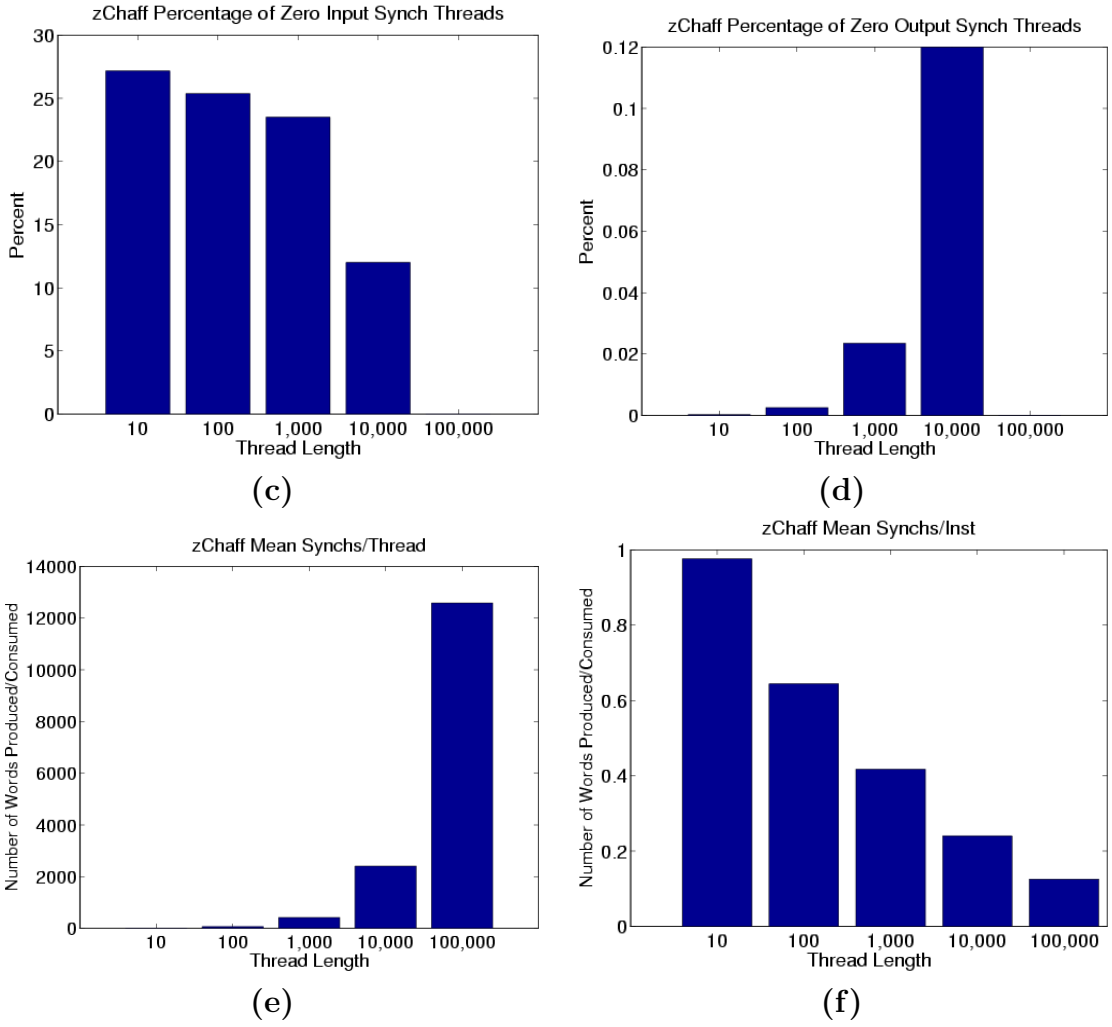


Figure C.14 (continued) zChaff Thread Properties (Continued on the next page.)

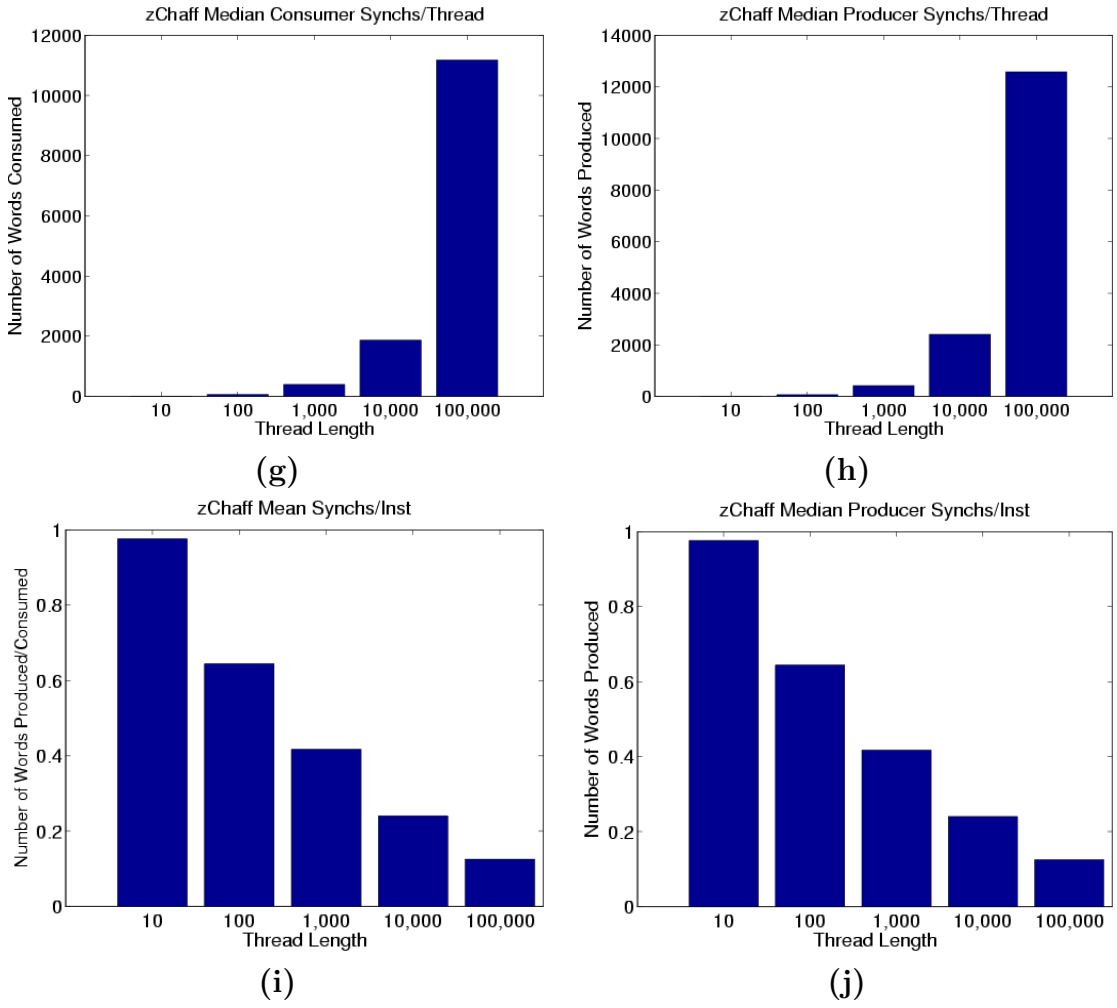
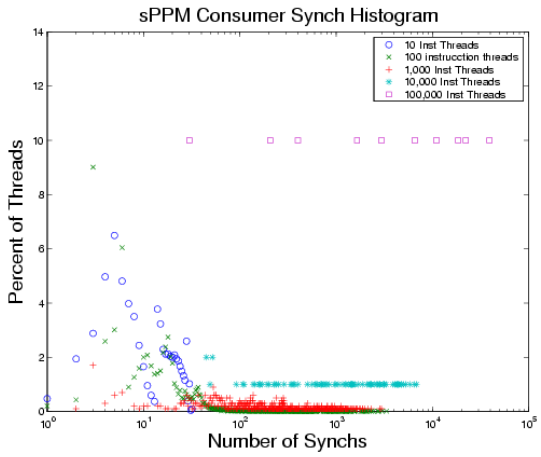
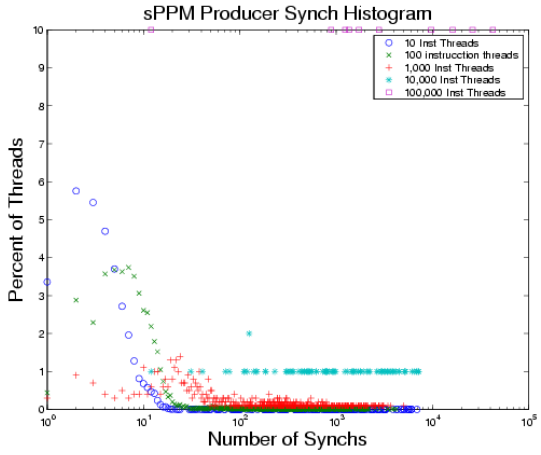


Figure C.14 (continued) zChaff Thread Properties



(a)



(b)

Figure C.15. sPPM Thread Properties (Continued on the next page.)

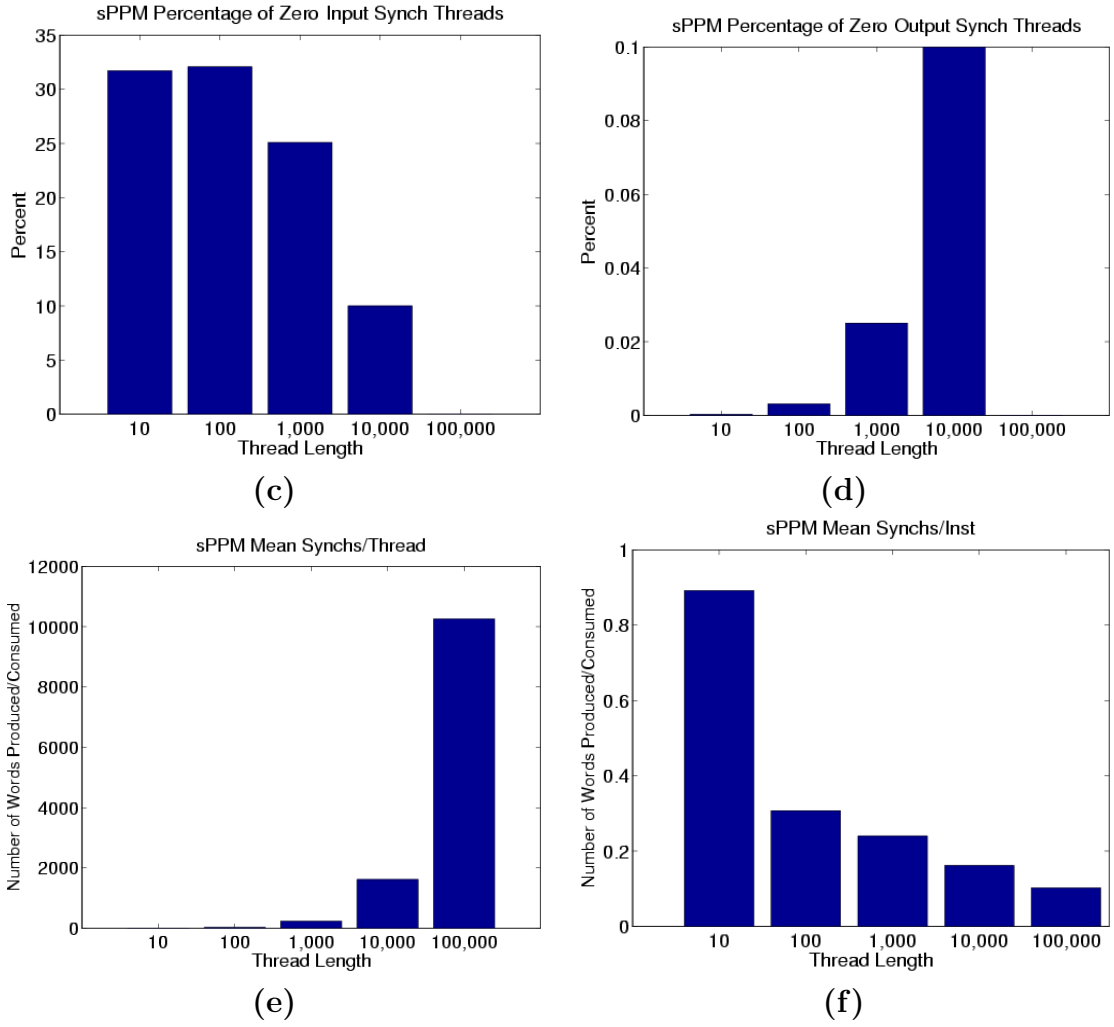


Figure C.15 (continued) sPPM Thread Properties (Continued on the next page.)

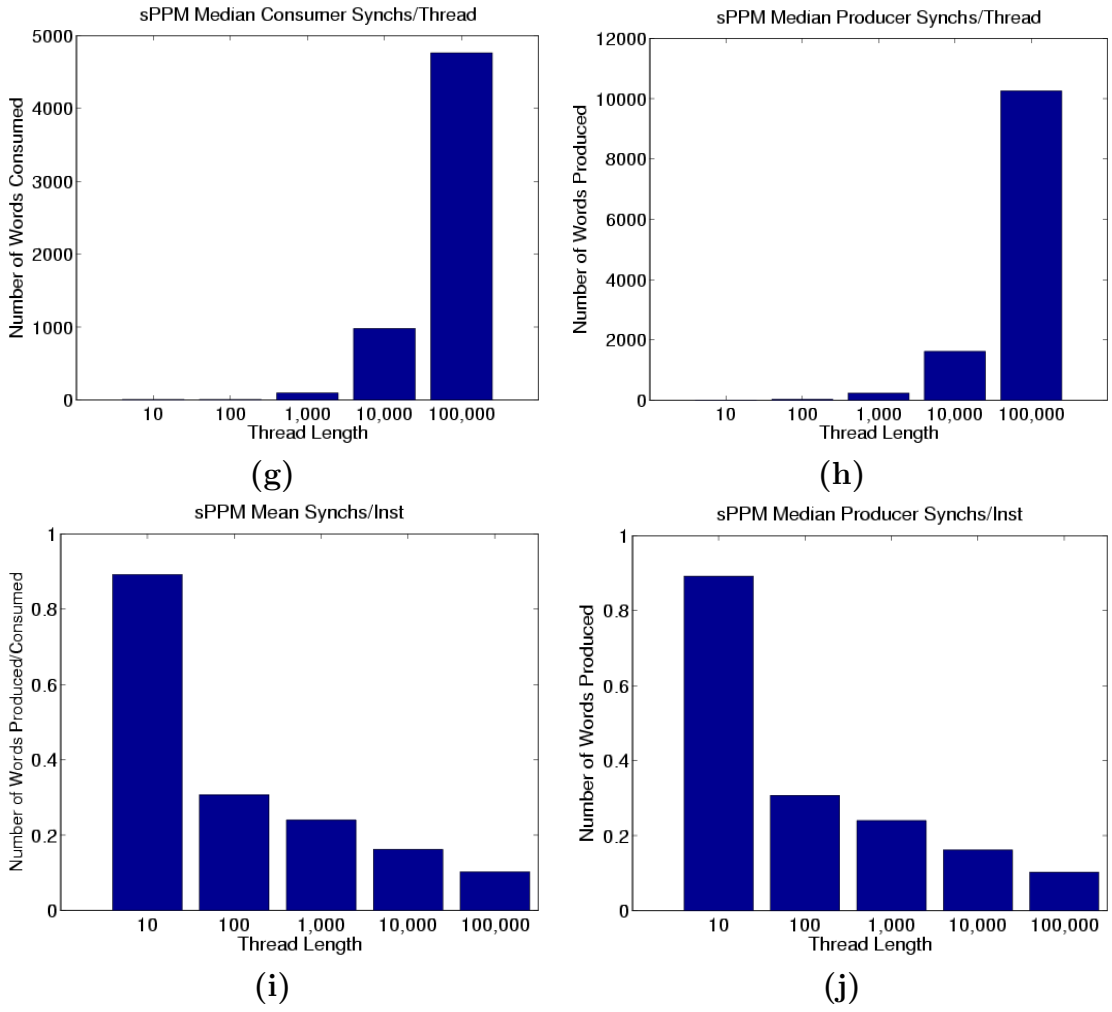


Figure C.15 (continued) sPPM Thread Properties

APPENDIX D

FULL DATA PARTITIONING RESULTS

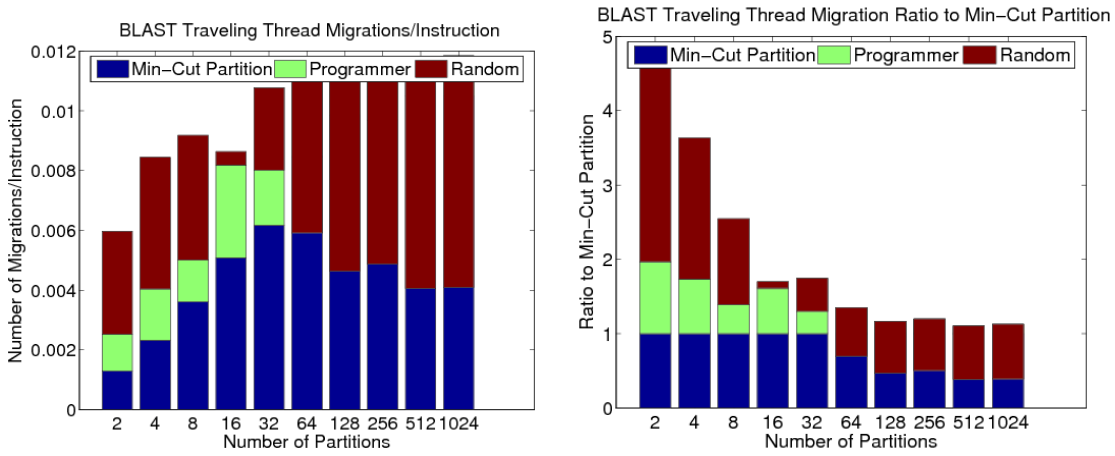


Figure D.1. BLAST Data Transition Graph Partitioning

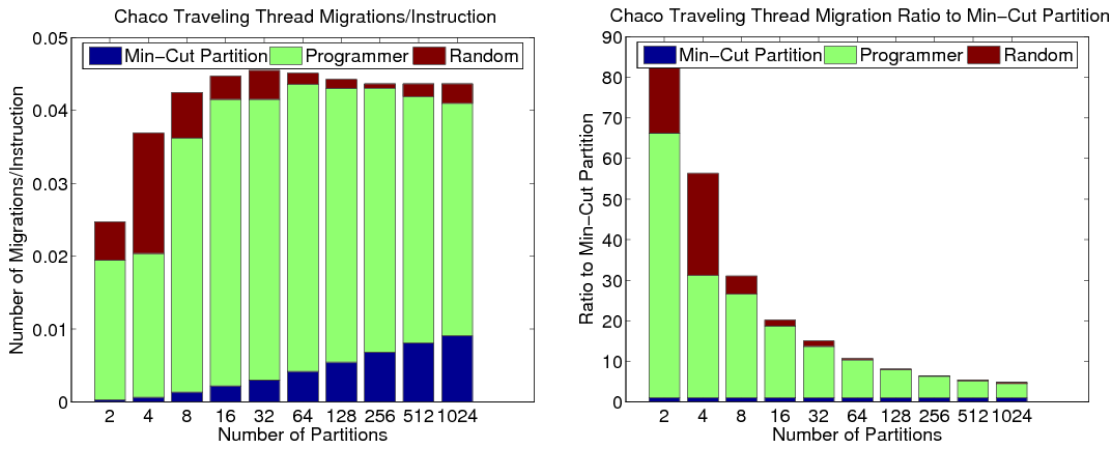


Figure D.2. Chaco Data Transition Graph Partitioning

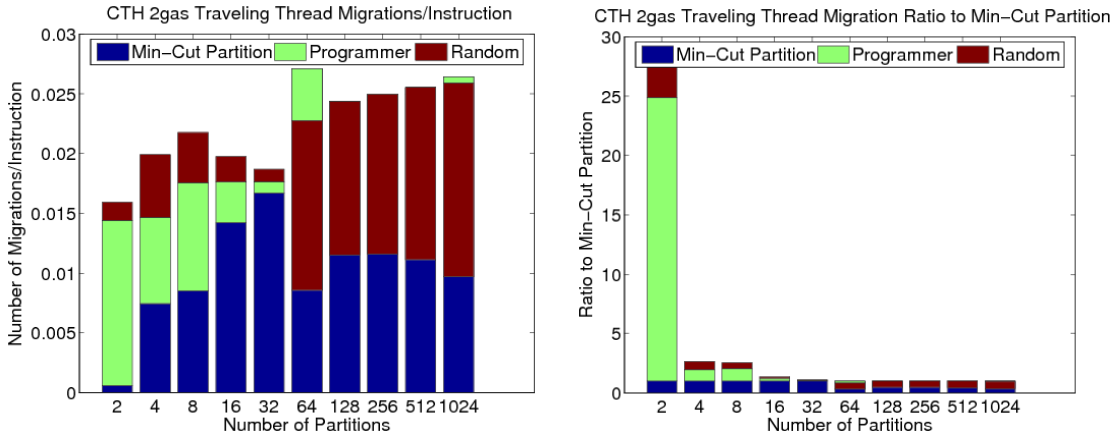


Figure D.3. CTH 2gas Data Transition Graph Partitioning

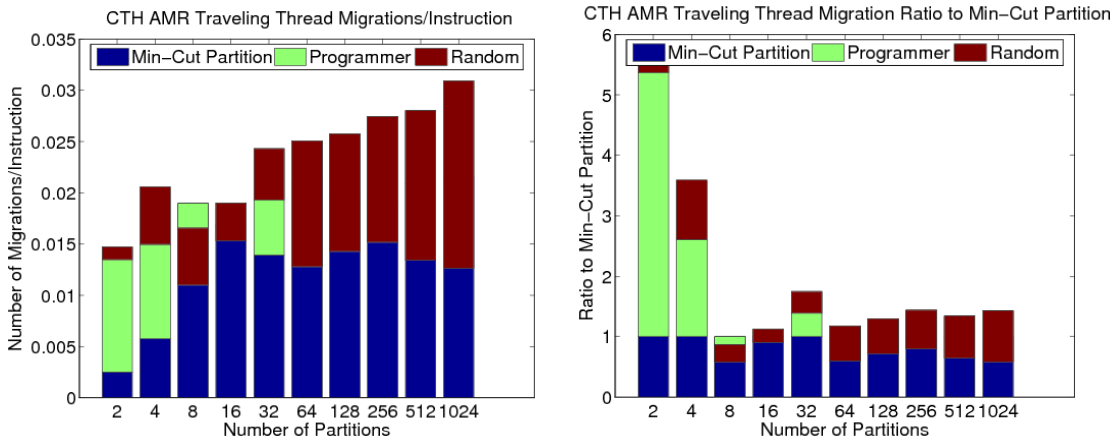


Figure D.4. CTH AMR Data Transition Graph Partitioning

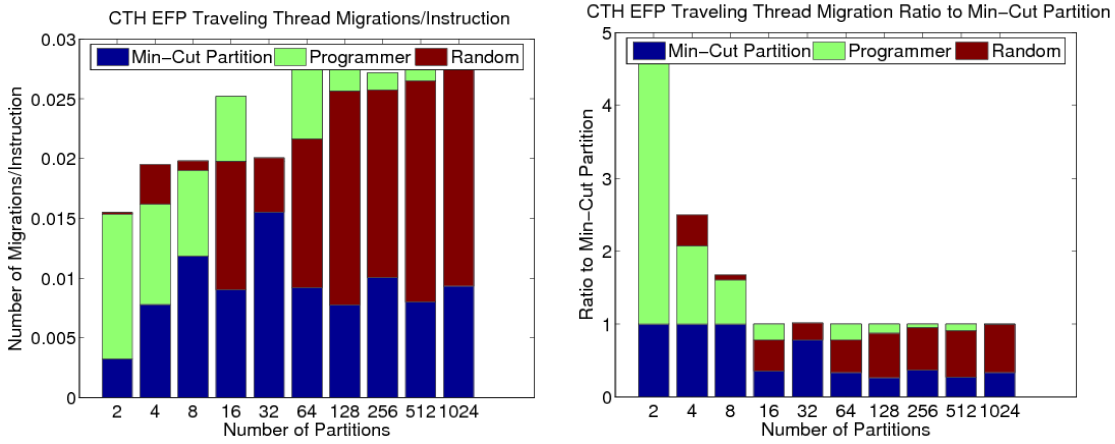


Figure D.5. CTH EFP Data Transition Graph Partitioning

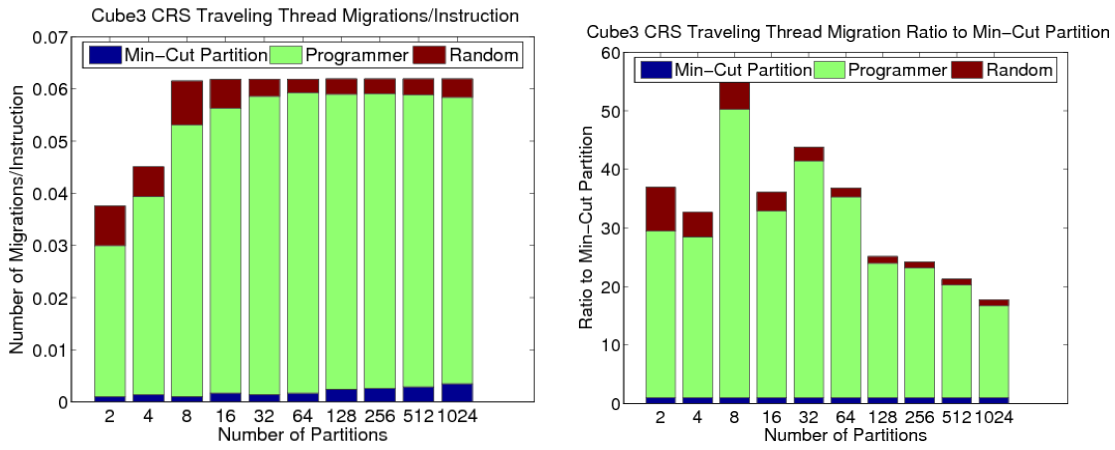


Figure D.6. Cube3 CRS Data Transition Graph Partitioning

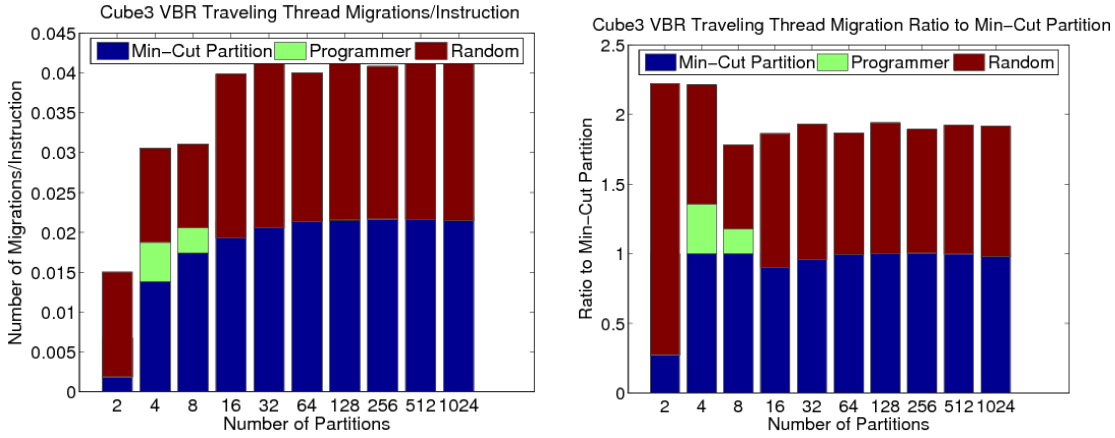


Figure D.7. Cube3 VBR Data Transition Graph Partitioning

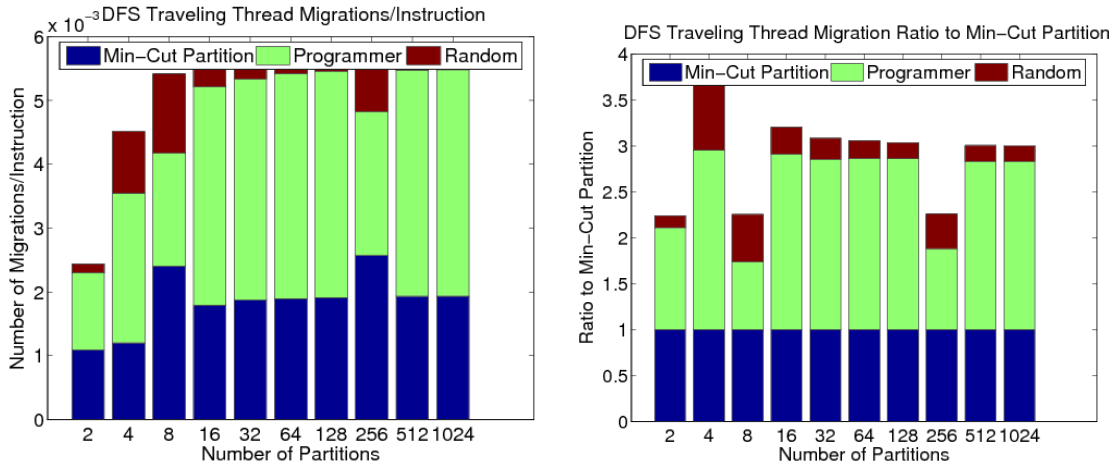


Figure D.8. DFS Data Transition Graph Partitioning

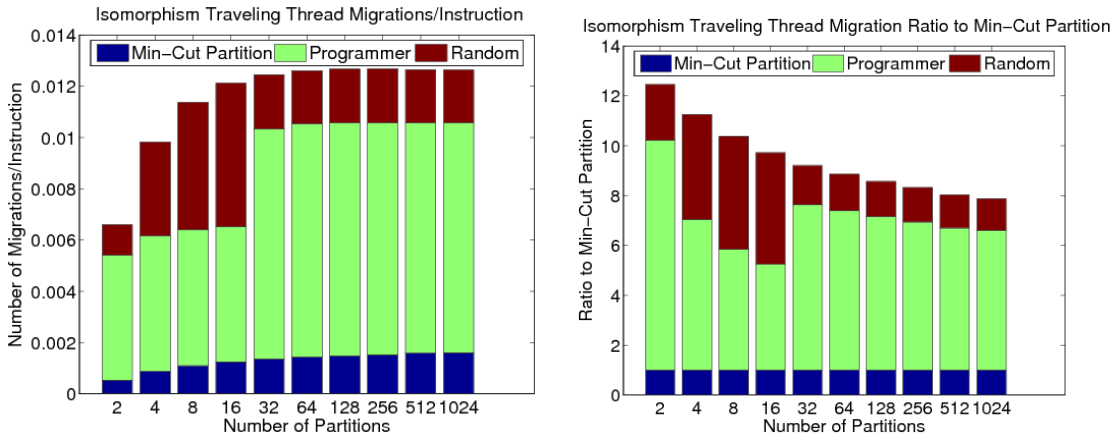


Figure D.9. Isomorphism Data Transition Graph Partitioning

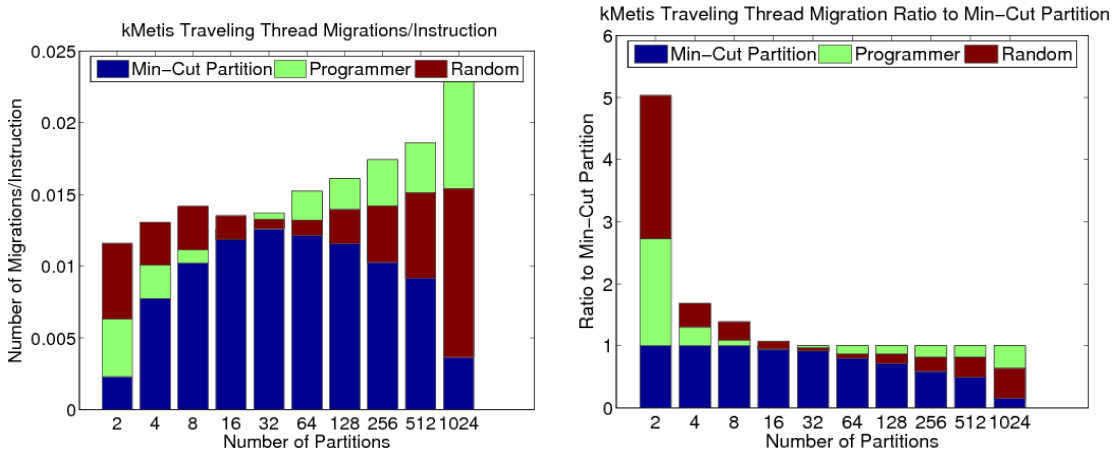


Figure D.10. kMetis Data Transition Graph Partitioning

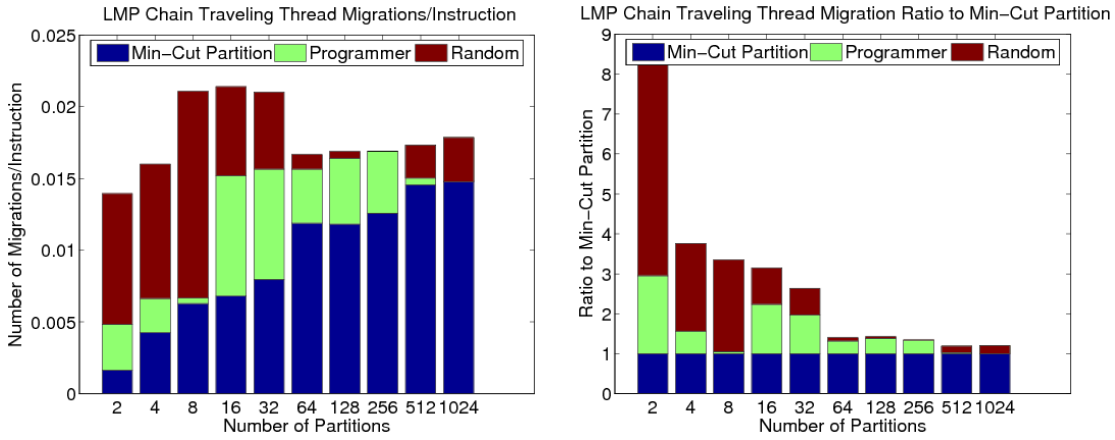


Figure D.11. LMP Chain Data Transition Graph Partitioning

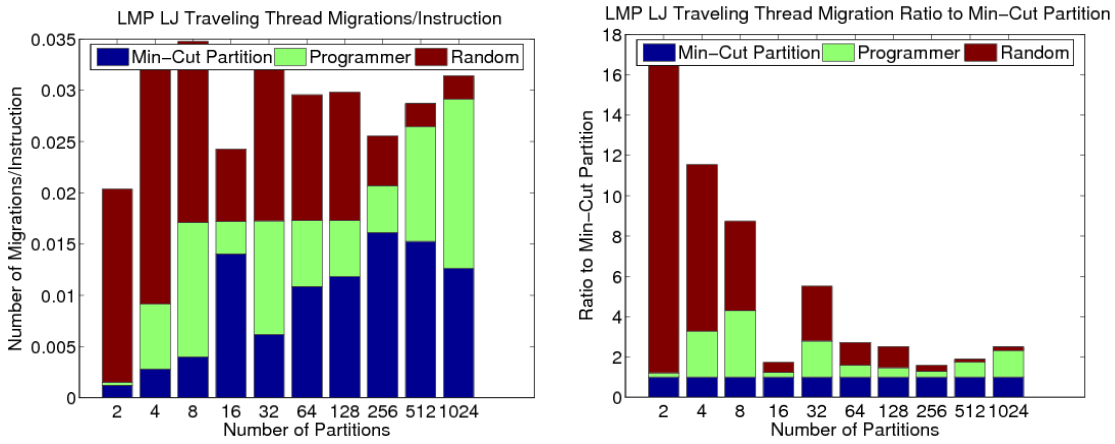


Figure D.12. LMP LJ Data Transition Graph Partitioning

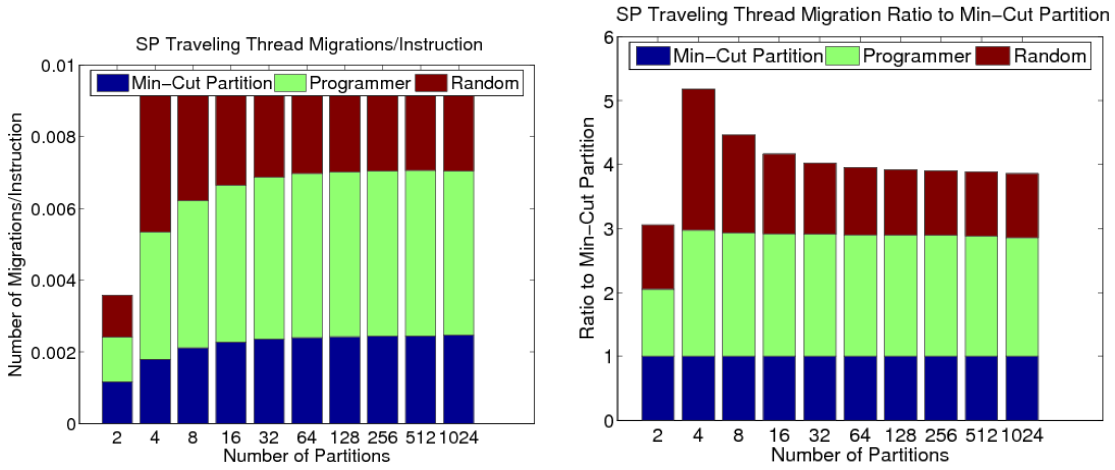


Figure D.13. SP Data Transition Graph Partitioning

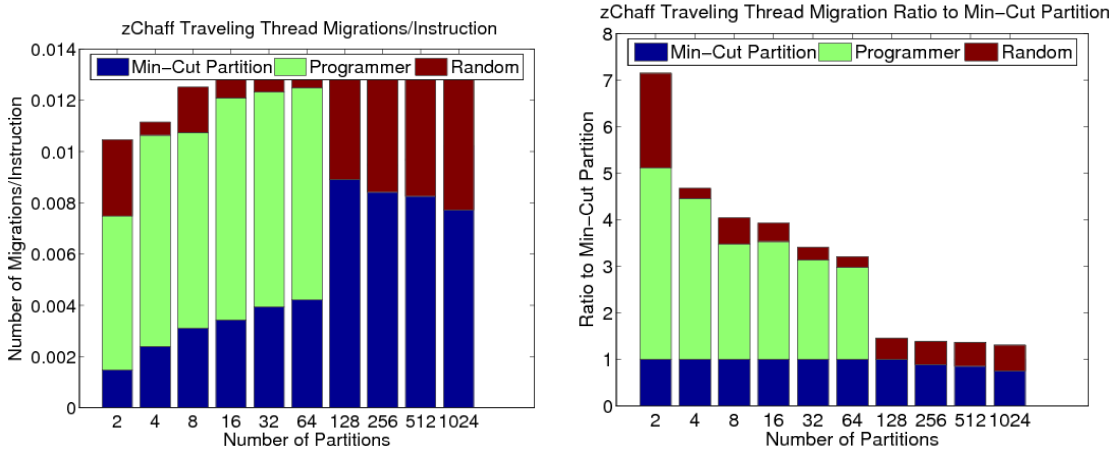


Figure D.14. zChaff Data Transition Graph Partitioning

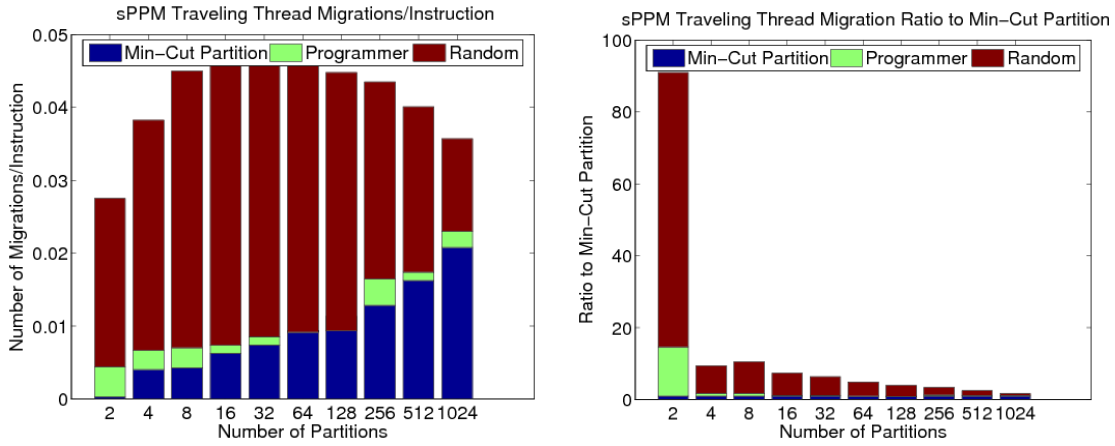


Figure D.15. sPPM Data Transition Graph Partitioning

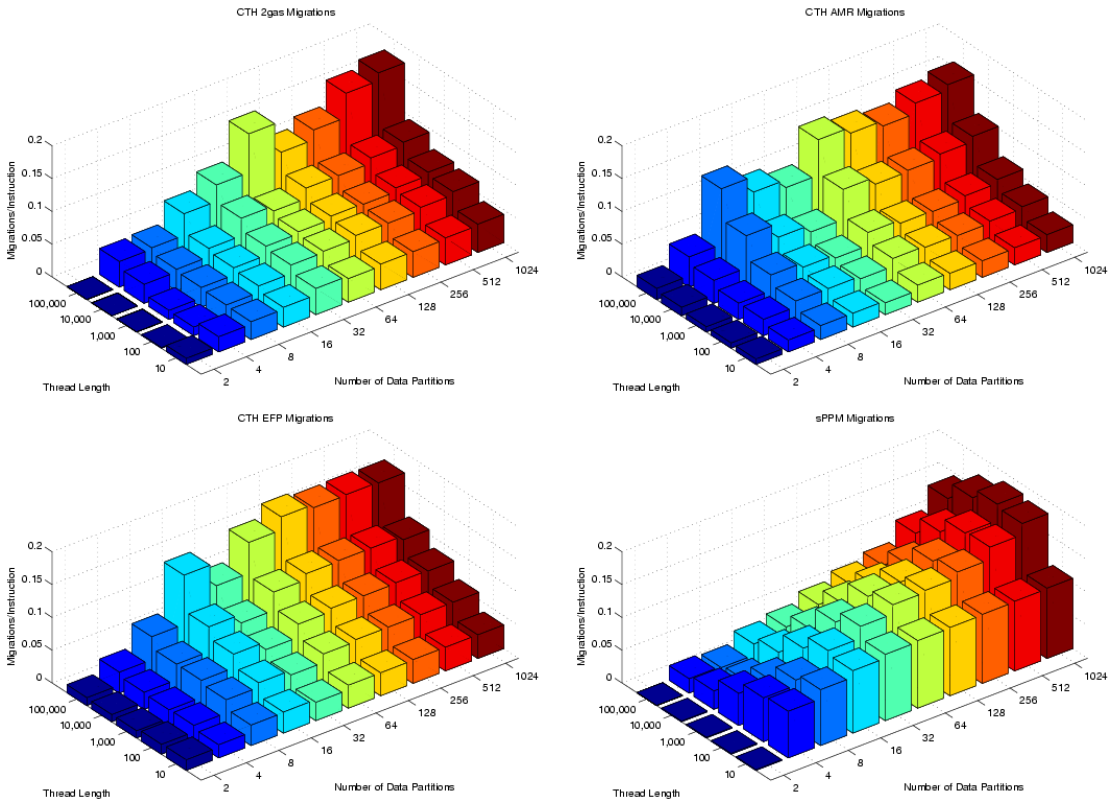


Figure D.16. Sandia Floating Point Suite Thread Migration. (Continued on the next page.)

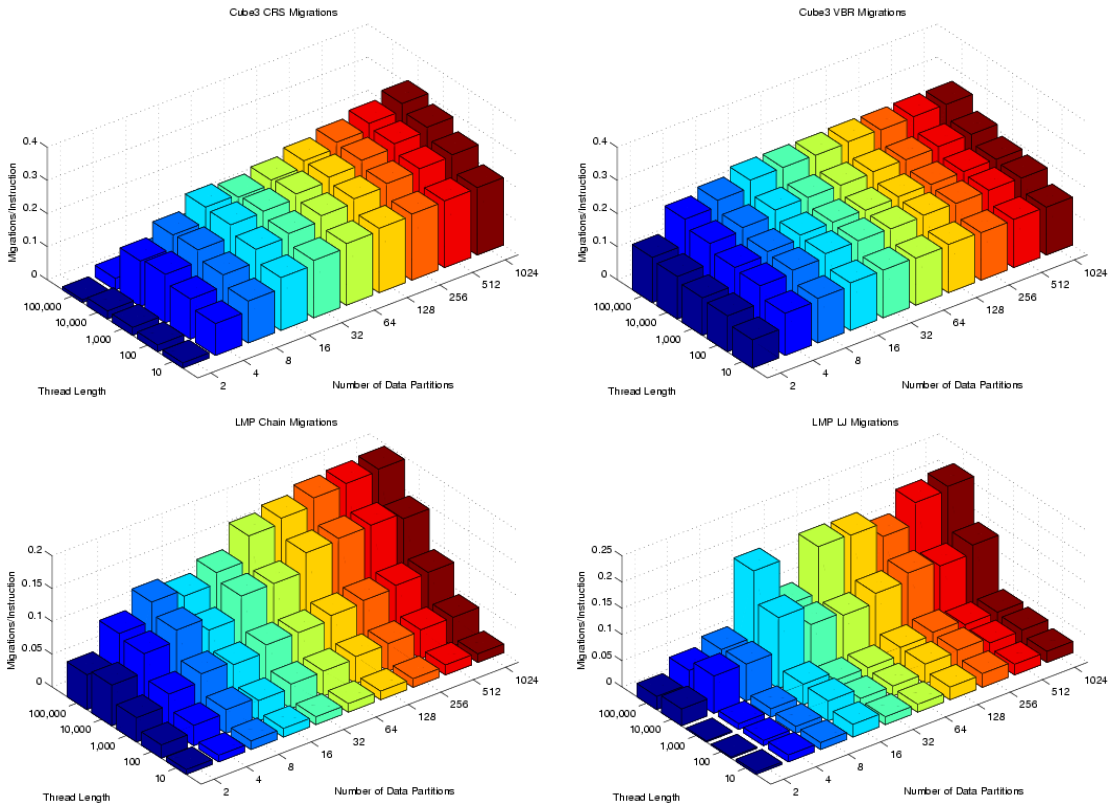


Figure D.16 (continued) Sandia Floating Point Suite Thread Migration.

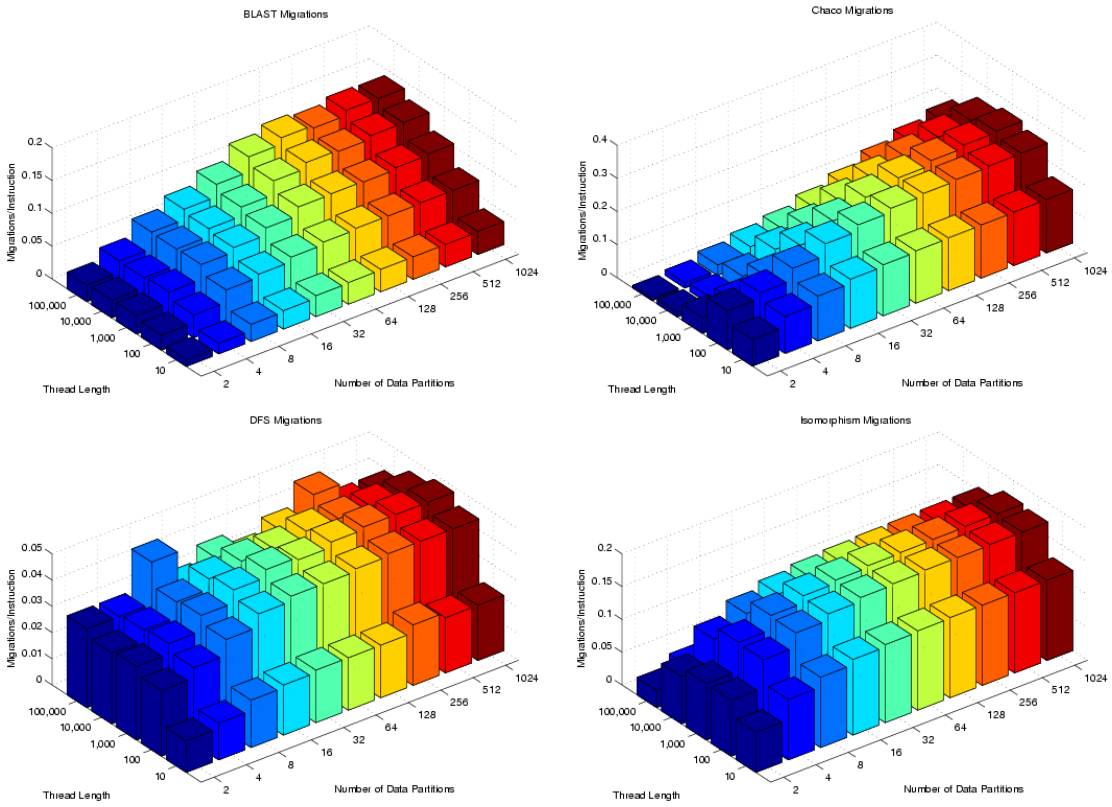


Figure D.17. Sandia Integer Suite Thread Migration. (Continued on the next page.)

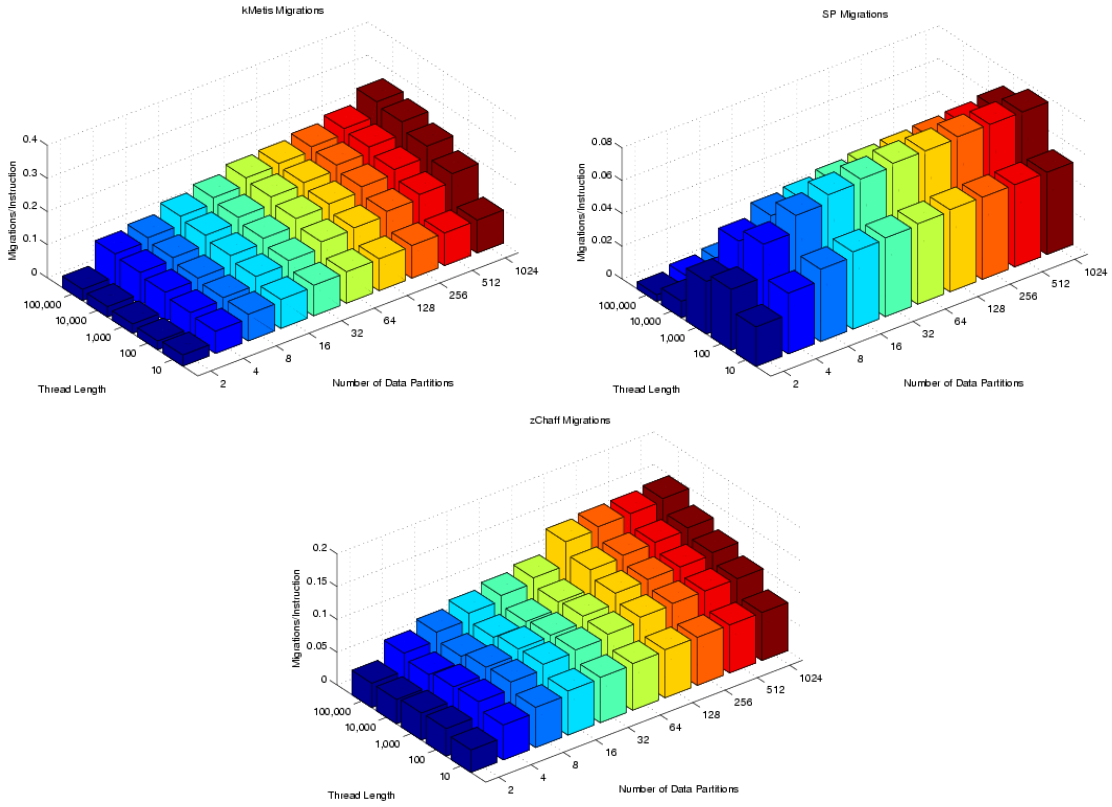


Figure D.17 (continued) Sandia Integer Suite Thread Migration.

BIBLIOGRAPHY

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] S.V. Adve and M.D. Hill. A unifed formalization of four shared memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[3] S.V. Adve and M.D. Hill. Weak ordering – a new definition. *Proceedings of the 17th Annual International Symposium on Computer Aruchtiecture*, pages 2–14, May 1990.

[4] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, , and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2–13, 1995.

[5] F. Allen, G. Almasi, W. Andreoni, D. Beece, B. J. Berne, A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, P. Coteus, P. Crumley, A. Curioni, M. Denneau, W. Donath, M. Eleftheriou, B. Fitch, B. Fleischer, C. J. Georgiou, R. Germain, M. Giampapa, D. Gresh, M. Gupta, R. Haring, H. Ho, P. Hochschild, S. Hummel, T. Jonas, D. Lieber, G. Martyna, K. Maturu, J. Moreira, D. Newns, M. Newton, R. Philhower, T. Picunko, J. Pitera, M. Pitman, R. Rand, A. Royyuru, V. Salapura, A. Sanomiya, R. Shah, Y. Sham, S. Singh, M. Snir, F. Suits, R. Swetz, W. C. Swope, N. Vishnumurthy, T. J. C. Ward, H. Warren, and R. Zhou. Blue Gene: A Vision for Protein Science Using a Petaflop Supercomputer. *IBM Systems Journal*, 4(2), 2001.

[6] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[7] Robert Alverson. Red Storm. In *Invited Talk, Hot Interconnects 10*, August 2003.

[8] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera System. *Tera Computer Company.*

[9] I. Amlani, A. Orlov, G. Toth, G.H. Bernstein, C.S. Lent, and G.L. Snider. Digital Logic Gates Using Quantum-Dot Cellular Automata. In *Science, 284:289-291*, 1999.

[10] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[11] Apple Architecture Performance Groups. *Computer Hardware Understanding Development Tools 2.0 Referenc e Guide for MacOS X*. Apple Computer Inc, July 2002.

[12] Atlantic Aerospace Electronics Corporation. *Data-Intensive Systems Benchmark Suite Analysis and Specification*, 1.0 edition, June 1999.

[13] John W. Backus. Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs. *CACM*, 21(8):613–641, 1978.

[14] Prithviraj Banerjee, John A. Chandy, Manish Gupta, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy, and Ernesto Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In *The First International Workshop on Parallel Processing*, pages 322–330, Bangalore, India, Dec. 1994.

[15] Randolph E. Bank, Todd F. Dupont, and Harry Yserentant. The hierarchical basis multigrid method. *Numerische Mathematik*, 52(4):427–458, 1988.

[16] R. Becker and R. Rannacher. A feed-back approach to error control in finite element methods: basic analysis and examples. *East-West J. Numer. Math.*, 4:237–264, 1996.

[17] Andrew D Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[18] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, November 1994.

[19] J.M. Borkenhagen, R.J. Eickemeyer, R.N. Kalla, and S.R. Kunkel. A Multithreaded PowerPC Processor for Commercial Servers. *IBM Journal of Research and Development*, 44(6):885–898, November 2000.

[20] Jay B. Brockman, Peter M. Kogge, Vincent Freeh, Shannon K. Kuntz, and Thomas Sterling. Microservers: A new memory semantics for massively parallel computing. In *International Conference on Supercomputing, Rhodes Greece*, pages 454–463, June 20-25, 1999.

[21] Joseph Tobin Buck. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Technical Report ERL-93-69, 1993.

[22] Doug Burger. System-Level Implications of Processor-Memory Integration. *Proceedings of the 24th International Symposium on Computer Architecture*, June, 1997.

[23] Doug Burger and Todd Austin. *The SimpleScalar Tool Set, Version 2.0*. SimpleScalar LLC.

[24] Doug Burger, James R. Goodman, and Alain Kagi. Limited Bandwidth to Affect Processor Design. *IEEE Micro*, 17(6):55–62, November/December 1997.

[25] Doug Burger and Alain Kagi. Memory Bandwidth Limitations of Future Microprocessors. *Proceedings of the 23th International Symposium on Computer Architecture*, May, 1996.

[26] A.W. Burks, H.H. Goldstine, and John von Neumann. Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. 1946.

[27] E. Caldwell, A. B. Kahng, and I. L. Markov. Design and Implementation of the Fiduccia-Mattheyses Heuristic for VLSI Netlist Partitioning. *Proceedings of the Workshop on Algorithm Engineering and Experimentation (ALENEX).*

[28] William J. Camp and James L. Tomkins. Thor's hammer: The first version of the Red Storm MPP architecture. In *In Proceedings of the SC 2002 Conference on High Performance Networking and Computing*, Baltimore, MD, November 2002.

[29] Michael J. Carey, David J Dewitt, and Jeffery F. Naughton. The oo7 benchmark. In *Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data*, 1993.

[30] John B. Carter, Wilson C. Hsieh, Leigh Stoller, Mark R. Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. Impulse: Building a smarter memory controller. In *HPCA*, pages 70–79, 1999.

[31] F. T. Chong, S. D. Sharma, E. A. Brewer, and J. Saltz. Multiprocessor runtime support for fine-grained, irregular DAGs. *Parallel Processing Letters*, 5(4):671–683, 1995.

[32] Daniel Citron, John Hennessy, David Patterson, and Gurindar Sohi. The use and abuse of SPEC: An ISCA panel. *IEEE Mico*, 23(4):73–77, July-August 2003.

[33] David Culler, Kim Keeton, Cedric Krumbein, Lok Tin Liu, Alan Mainwaring, Rich Martin, Steve Rodrigues, Kristin Wright, and Chad Yoshikawa. Generic Active Message Interface Specification. February 1995.

[34] David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, 1991.

[35] W.J. Dally, J. Fiske, J Keene, R. Lethin, M. Noakes, P. Nuth, R. Davidson, and G. Fyler. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, April 1992.

[36] Stefanos Damianakis, Angelos Bilas, Cezary Dubnicki, and Edward W. Felten. Client Server Computing on the SHRIMP Multicomputer. *IEEE Micro*, February 1997.

[37] Michael D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Redmond, WA, USA, 26 July 1994.

[38] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *Technical Report No. CS-93-214 (revised), University of Tennessee*, April 1994.

[39] Ian Foster, Ming Xu, Bhaven Avalani, and Alok Choudhary. A Compilation System That Integrates High Performance Fortran and Fortran M. In *Proc. 1994 Scalable High Performance Computing Conf.*, page July. IEEE Computer Science Press, 94.

[40] Guang R. Gao and Vivek Sarkar. Location consistency: Stepping beyond the memory coherence barrier. *Proceedings of the 24th International Conference on Parallel Processing*, August 14-18, 1995.

[41] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[42] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[43] Roberto Grosso, Christoph Lürig, and Thomas Ertl. The multilevel finite element method for adaptive mesh optimization and visualization of volume data. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization '97*, pages 387–394, 1997.

[44] Manish Gupta and Prithviraj Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):179–193, 1992.

[45] S. Ha, H. Kim, and S. Han. The efficient implementation of sequential loops in multithreaded computation, 1995.

[46] J. M. Haile. *Molecular Dynamics Simulation : Elementary Methods*. John Wiley & Sons, 1997.

[47] Michael Halbherr, Yuli Zhou, and Christopher F. Joerg. MIMD-style parallel programming based on continuation-passing thread. Technical Report CSG Memo-335, 1994.

[48] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Apoorv Srivastava, William Athas, Jay Brockman, Vincent Freeh, Joonseok Park, and Jaewook Shin. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. In *Supercomputing, Portland, OR*, November 1999.

[49] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics.

[50] B. Hendrickson and R. Leland. The chaco user's guide — version 2.0. Technical Report SAND94-2692, 1994., 1994.

[51] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, 2002.

[52] Michael Heroux, Roscoe Bartlett, Vicki Howle, Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. *Sandia National Labs, SAND REPORT 2003-2927*, August 2003.

[53] James Hicks, Derek Chiou, Boon Seong Ang, and Arvind. Performance studies of Id on the Monsoon Dataflow System. *Journal of Parallel and Distributed Computing*, 18(3):273–300, 1993.

[54] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex., 1993.

[55] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23, pages 35–46, Atlanta, GA, June 1988.

[56] Robert A. Iannucci, Guang R. Gao, and Jr. Robert H. Halstead. *Multithreaded Computer Architecture: A Summary of the State of the Art*. Kluwer, August 1994.

[57] IEEE Standard 1596-1992. The SCI Standard.

[58] Jagannathan. Dataflow models. In *Parallel and Distributed Computing Handbook, McGraw-Hill, 1996*. 1996.

[59] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[60] Peter M. Kogge. *The Architecture of Symbolic Computers*. McGraw Hill, 1991.

[61] Peter M. Kogge, Jay B. Brockman, and Vincent Freeh. Processing-In-Memory Based Systems: Performance Evaluation Considerations. In *Workshop on Performance Analysis and its Impact on Design held in conjunction with ISCA, Barcelona, Spain*, June 27-28, 1998.

[62] Peter M. Kogge, Jay B. Brockman, and Vincent W. Freeh. PIM Architectures to Support Petaflops Level Computation in the HTMT Machine. In *3rd International Workshop on Innovative Architectures, Maui High Performance Computer Center, Maui, HI*, November 1-3, 1999.

[63] Kogge, Peter M. and et al. *Final Report: PIM Architecture Design and Supporting Trade Studies for the HTMT Project*, September 1999.

[64] Kogge, Peter M. and et al. *DIVA Final Report*, 2001.

[65] J. T. Kuehn and B. J. Smith. The Horizon Supercomputer System: Architecture and Software. *Proceedings of Supercomputing 1988*, November 1988.

[66] Shannon K. Kuntz, Richard C. Murphy, Michael T. Niemier, Jesus Izaguirre, and Peter M. Kogge. Petaflop Computing for Protein Folding. In *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, VA*, March 12-14, 2001.

[67] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, , and John Hennessy. The Stanford FLASH Multiprocessor. *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[68] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Nineteenth International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, 1992. ACM and IEEE Computer Society.

[69] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Implementation and Performance. *19th International Symposium on Computer Architecture*, 1993.

[70] C.S. Lent. Molecular Electronics: Bypassing the Transistor Paradigm. In *Science, 288:1597-1599*, 2000.

[71] C.S. Lent and P.D. Touugaw. A Device Architecture for Computing with Quantum Dots. In *Proceedings of the IEEE, 85:541*, 1997.

[72] Arthur B. Maccabe, Kevin S. McCurley, Rolf Riesen, and Stephen R. Wheat. SUNMOS for the Intel Paragon: A Brief User's Guide. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference*, pages 245–251, June 1994.

[73] Arthur B. Maccabe, Rolf Riesen, and David W. van Dresser. Dynamic processor modes in Puma. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 8(2):4–12, 1996.

[74] McCalpin, John D. *Stream: Sustainable memory bandwidth in high performance computers*, 1997.

[75] Simon W. Moore. *Multithreaded Processor Design.* Kluwer Academic Publishers, June 1996.

[76] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[77] Richard C. Murphy. *Design Parameters for Distributed PIM Memory Thesis.* MS CSE Thesis, University of Notre Dame, April 2000.

[78] Richard C. Murphy and Peter M. Kogge. Trading Bandwidth for Latency: Managing Continuations Through a Carpet Bag Cache. In *Proceedings of the International Workshop on Innovative Architecture 2002 (IWIA02)*. IEEE Computer Society, January 10-11, 2002.

[79] Richard C. Murphy, Peter M. Kogge, and Arun Arun Rodrigues. The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems. In *Proceedings of the Second Workshop on Intelligent Memory Systems, held in conjunction with ASPLOS-IX, Cambridge, MA*. ACM Press, November 12-15, 2000.

[80] Walid A. Najjar, A. P. Wim Bohm, and W. Marcus Miller. A quantitative analysis of dataflow program execution — preliminaries to a hybrid design. *Journal of Parallel and Distributed Computing*, 18(3):314–326, 1993.

[81] Lakshmi Nanayanaswamy and Peter M. Kogge. Combinators-In-Memory: An Unconventional Approach to Avoiding the Memory Wall. *1st Int. Conf. on Unconventional Models of Computation, Auckland, NZ*, Jan. 5-8, 1998.

[82] Next. *Nextstep Object-Oriented Programming and the Objective C Language: Release 3*. Addison Wesley, 1993.

[83] SPEC Open Systems Steering Committee. Spec run and reporting rules for cpu95 suites. September 11, 1994.

[84] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active Pages: A Computation Model for Intelligent Memory. *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[85] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, volume 19, pages 342–351, New York, NY, 1991. ACM Press.

[86] David Patterson, Thomans Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, April, 1997.

[87] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface, 2ed.* Morgan Kaufmann Publishers, 1997.

[88] John R. Pilkington and Scott B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves:. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):288–300 (or 288–299??), 1996.

[89] Steven J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal Computation Physics*, 117:1–19, 1995.

[90] Steven J. Plimpton, R. Pollock, and M. Stevens. Particle-mesh ewald and rRESPA for parallel molecular dynamics. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.

[91] Ravi Ponnusamy, Joel H. Saltz, Alok N. Choudhary, Yuan-Shin Hwang, and Geoffrey Fox. Runtime support and compilation methods for user-specified irregular data distributions. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):815–831, 1995.

[92] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, 1991.

[93] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Supercomputing '89*, November 1989.

[94] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6 (part 2)), 1978.

[95] Arun F. Rodrigues. *Programming Future Architectures: Dusty Decks, Memory Walls, and the Speed of Light*. Ph.D. Dissertation, University of Notre Dame, 2006.

[96] Richard M. Russell. The CRAY-1 Computer System. *CACM*, 21(1):63–72, January 1978.

[97] Ashley Saulsbury, Tim Wilkinson, John Carter, and Anders Landin. An Argument For Simple COMA. *First IEEE Symposium on High Performance Computer Architecture*, January 1995.

[98] Lance Shuler, Chu Jong, Rolf Riesen, David van Dresser, Arthur B. Maccabe, Lee Ann Fisk, and T. Mack Stallcup. The Puma operating system for massively parallel computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.

[99] T. Sterling and L. Bergman. A design analysis of a hybrid technology multithreaded architecture for petaflops scale computation. In *International Conference on Supercomputing, Rhodes, Greece*, June 20-25, 1999.

[100] Sun Microsystems. *Introduction to Shade*, June 1997.

[101] M. R. Thistle and B. J. Smith. A Processor Architecture for Horizon. *Proceedings of Supercomputing 1988*, November 1988.

[102] Stephen R. Wheat Timothy G. Mattson, David Scott. A TeraFLOPS Supercomputer in 1996: The ASCI TFLOP System. In *Proceedings of the 1996 International Parallel Processing Symposium*, 1996.

[103] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[104] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, January 1967.

[105] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*. ACM Press, 1992.

[106] David H.D. Warren and Seif Haridi. The Data Diffusion Machine – A Scalable Shared Virtual Memory Multiprocessor. *1988 International Conference on Fifth Generation Computer Systems*.

[107] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[108] Ruhong Zhou, Edward Harder, Huafeng Xu, and B.J. Berne. Efficient Multiple Time Step Method for use with Ewald and Particle Mesh Ewald for Large Biomolecular Systems. *The Journal of Chemical Physics*, 115, 2001.