Memory Management in a Massively Parallel Shared-Memory Environment

A Proposal

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

Ph.D.

by

Kyle B. Wheeler, B.S., M.S.C.S.E.

Douglas Thain, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
April 2007

Memory Management in a Massively Parallel Shared-Memory Environment

Abstract

by

Kyle B. Wheeler

This paper discusses the implementation of a scalable runtime for massively parallel computers based on processor-in-memory (PIM) technology. Of particular importance for high-performance computing is the ability to scale the runtime down to the minimum necessary overhead for a given computation. Because of the PIM architecture, virtually every fundamental operation in such a system can be described as a memory operation. This characteristic makes tasks such as data layout, distribution, and relocation fundamental to the operation and performance of applications. This paper proposes research into the effects of data layout decisions that can be made at runtime either explicitly or as an adaptive optimization, and how such a system may be able to recover from poor decisions.

CONTENTS

FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Computers throughout history have been plagued by the same problem: the need to function faster. This is a problem with as many facets as there are researchers to study them, and changes with every new solution that is found, however the standard approach is to find a bottleneck in the computer system and remove it.

Modern supercomputers have largely taken the route of parallel computation to achieve increased computational power [3, 4, 9, 18, 28, 38, 78]. In many modern supercomputers, shared memory has been traded for distributed memory because of the cost and complexity involved in scaling shared memory to the size systems needed for supercomputing applications. This design is vulnerable to an obvious bottleneck: the communication between parallel nodes.

Since the dawn of computing, the most persistent bottleneck in performance has been latency: the processor is faster than the storage mechanism and must wait for it to return data to be used in computation. Whether the disparity is high-speed vacuum tubes waiting for high-density (low speed) drum memory, or high-speed CPU's waiting for high-density (low speed) DRAM, the problem is the same. This is a fundamental vulnerability in the von Neumann computational model,

because of the separation of computation from storage and their dependence on one another. Computation can only proceed at the pace of the slower of the two halves of the computational model. The slower has historically been the memory, so it is generally termed "memory latency" or the "memory wall". This problem negatively impacts large data-dependent computation particularly strongly [43, 67, 80].

There are only two fundamental approaches to addressing this problem of memory latency: either avoid the problem or tolerate the problem. The "toleration" approach focuses on the overall throughput of the system, and masks the latency of requesting things from memory by performing unrelated useful work while waiting for each memory operation to complete. An example of this idea is the work in Simultaneous Multithreading (SMT) [45, 62] and out-of-order execution, both of which identify work that is sufficiently unrelated to be performed at the same time. The fundamental characteristic of computation that allows this approach to work is concurrency: the more unrelated work that needs to be accomplished, the greater the ability to do something else while memory operations complete.

The other approach, "avoidance", generally uses specialized hardware to prevent latency effects. A common example of such specialized hardware is a memory cache, which provides a small and very fast copy of the slower high-density memory. The fundamental characteristic of computation that allows caches to work well is locality: the smaller the set of data that must be used at a time for computation, the more it can be stored in a small, fast cache. As caches are smaller than main memory, and generally smaller than most data sets used in computation, they do not solve the problem, but do avoid it to some degree.

Von Neumann himself recognized the problem of latency and proposed the cache-based avoidance technique in a quotation from 1946 [13] that is very commonly cited in computer architecture texts [25, 56]:

> 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.

As predicted, this bottleneck has been a powerful consideration in computer design and architectural research for the past six decades, with each new computer architecture seeking to address the problem in one way or another. Despite this effort, the problem has only gotten worse with time. Figure 1.1 depicts the progress of the problem in simple terms, based on information from Intel's processor line, starting in 1971 with the 4004 and continuing through the Pentium IV in 2006.

One way of addressing this problem is to integrate processors with memory: create a system composed of small processors attached directly to memory. This idea has been examined by projects such as the EXECUBE [36] and its processor-in-memory (PIM) successors [9, 10, 37, 48], IRAM [55], Raw [79], Smart Memories [42], Imagine [61], FlexRAM [33], Active Pages [53], DIVA [23], Mitsubishi's M32R/D [51], and NeoMagic's MagicGraph [50] chips, to name a few. The major attraction of integrating memory and processor is that it brings the data much closer to where it will be used, and thus provides the opportunity for both increasing bandwidth and decreasing latency.

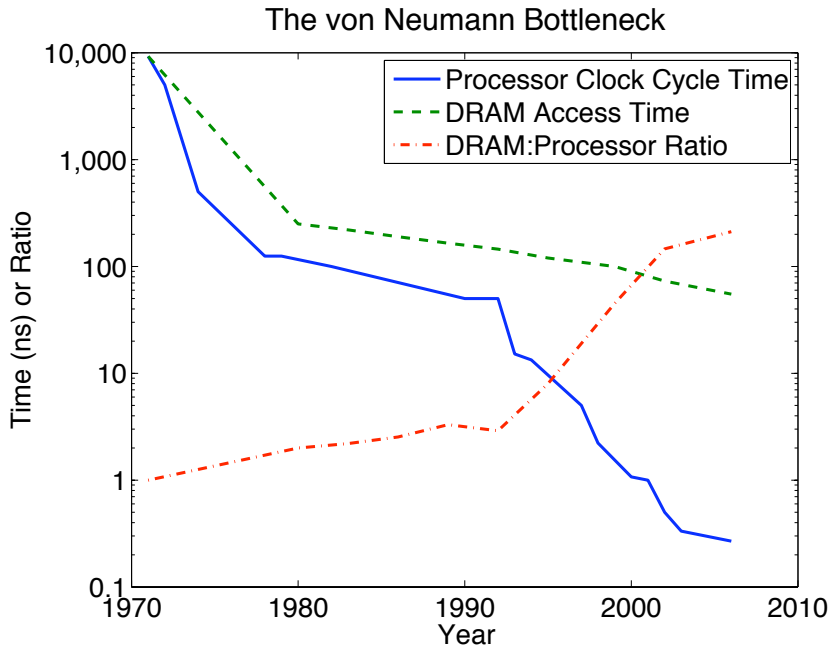

Figure 1.1. The von Neumann Bottleneck. [47]

It is conceivable to build a shared-memory parallel computer almost entirely out of PIM units. Such a computer would have an extremely large number of parallel computational nodes, however each node would not be as fast or as complex as a modern central processor. This unusual architecture would place unusual demands on the software that runs on it and on the developers who write that software, in large part because the standard issues of scheduling and data layout are combined.

Typical operating systems in the supercomputing field are monolithic kernels [34, 65, 68] because of the speed advantages typically found in monolithic design. However, such kernels do not typically lend themselves to internal parallelization or wide scale modularization. Along monolithic lines, each node in a large scale PIM-based shared memory system could be treated as a separate system, in

need of its own operating system or its own monolithic kernel. Such an approach may appear similar to a standard distributed operating system, but because of the shared address space is more closely related to the virtual-node approach in some high-performance operating systems, such as FastOS [34, 68]. FastOS can treat the multiple processing cores within shared-memory nodes—such as dual-core Opterons—as distinct nodes in the larger system. However, logically separating each PIM processor into an independent processing node does not exploit the advantages offered by such a large, fast, shared memory system designed for highly threaded applications [9, 10]. The benefit of a shared memory parallel system is that memory operations can be done relatively quickly by any node in the system.

A common design for parallel supercomputers is to divide the component nodes of the computer into specialized groups, as such specialization can provide significant performance improvements [52]. This can be accomplished even in shared memory machines, but because the address space is shared, the specialization can happen at a more fine-grained level. For example, operating system operations such as memory management for a cluster of nodes within the system can be offloaded to a single node in that cluster [44]. Servicing system calls, or doing garbage collection, or even the storage of the implementations of less-used OS features could also be offloaded to specialized processors in such a tightly-coupled parallel environment.

1.2 Problem

Caching is one of the most well-known and well-studied ways of addressing the von Neumann bottleneck. As such, it has drastically affected the design

of software and compilers by providing a simple metric for optimizing performance: locality. Modern compiler technology is designed to produce code that exploits commodity processor cache characteristics. Memory "hot spots" are created intentionally so that frequently used data will fit within the small amount of fast memory available. While this has proved extremely successful in simple von Neumann single-processor designs, supporting coherent caches in parallel systems places a significant burden on memory bandwidth [19]. Worse, the more nodes in the parallel system, the more expensive this cache coherency overhead becomes. The result is that rather than sharing data implicitly and at a fine-grained level, data is shared explicitly and large chunks of it at a time [22], so as to avoid the shared cache overhead. This leads some supercomputer designers to do away with processor cache entirely in large parallel computers [1], a design decision which invalidates the wisdom of creating hot-spots. However, because of the physical reality that in a large enough system not all memory addresses may be accessed with equal speed, randomly distributing data across all memory banks [1] does not take advantage of what ranks of locality naturally exist in hardware design. Additionally, this decision removes a major tool for addressing the problem of the memory wall.

The PIM design concept attacks the von Neumann bottleneck directly; by moving the processor closer to the memory—avoiding cache, bus, and memory-controller overhead—the latency of accessing memory is decreased. The use of many small processors rather than a single central processor also increases the potential for parallel execution, which improves tolerance of memory latency as well. Because each processor in a PIM-like system is attached directly to—and can thus access more quickly—a specific subset of all of the memory in the system,

it is most naturally described as a non-uniform memory access (NUMA) system.

In a NUMA system as potentially dynamic as a PIM-based shared memory system, placement of data becomes not merely a crucial problem, but a problem for which the optimal answer may change over the course of a given set of computations. More specifically, the problem is thus:

> *What layout is most beneficial to computation, what characteristics should be used in analyzing a given layout, and is it beneficial to modify that layout at runtime in an application-independent way?*

1.3 Contribution

The area of data-layout is a well-researched area [6], beginning even before memory hierarchies were first introduced in the early 1950's [64]. At the time, programmers needed to manage the memory hierarchies themselves, explicitly copying data from drum to core memory and back. In 1962, Kilburn et al. [35] proposed automatic management of core memory, which was first implemented in the Atlas computer. Similarly, as a skilled parallel programmer may well be able to plan data layout for large programs and manage it to provide optimal performance on any size system, it is both more convenient and more portable to do this automatically, both as is currently done at compile time and as this paper proposes, at runtime. While runtime reorganization was proposed and demonstrated by Ding et al. [17], the concept has not been applied to a parallel system. In addition to convenience and portability, runtime reorganization also reduces the amount of detailed knowledge of the system that a programmer must know to exploit the memory distribution of any particular large parallel system. The contribution of the proposed research will be first a metric for comparing layouts

based on memory access traces, and second, a comparison of several fundamental runtime layout modification techniques to gauge their utility and effect on several representative large scientific codes.

1.4 Prior Work

Work in memory layout generally attempts to optimize for one of two things: cache efficiency, or parallelism. Cache efficiency work typically relies on the reuse-distance metric—the number of instructions between uses of a given piece of data—to determine the success of the layout [75, 76], which biases data layout toward the creation of hot-spots [16, 20]. By contrast, parallel layout optimization [12, 15, 40] is based upon the idea that hot-spots prevent multiple nodes from efficiently accessing data concurrently, and thus biases data layout away from the creation of hot-spots. For example, Olivier Temam developed a compile-time technique for eliminating cache conflicts (hot-spots) through the use of data copying [77].

In a NUMA-based cache-free shared memory parallel system, such as a PIM-based system, both goals apply. Data needs to be kept local to a particular thread because nearby memory is faster than more distant memory, and yet data must also be kept far apart to be accessed quickly in parallel. Because these goals are inherently opposed, and because different applications can take advantage of different amounts of parallelism, they must be balanced in a way that is particular to the task at hand. Byoungro So, et al. demonstrated the potential of customized data layout algorithms for different applications [71]. Ding, et al. demonstrated that runtime modifications to data layout can provide significant improvements in application performance, particularly when the application's memory behavior

changes over time [17].

In shared memory systems, the cost of communication for remote memory references is the single largest factor in application performance [29, 49, 57, 65, 69, 70]. Typical shared-memory operating systems are designed for systems of between two and several hundred nodes, due to the difficulty of scaling to much larger numbers of nodes; large-scale shared memory systems are considered those with hundreds of nodes [30, 32]. Larger systems that behave like shared memory systems are usually implemented as distributed memory systems that emulate real shared memory with a virtual memory layer [21, 41]. Such emulation involves overhead [24] that is avoided when the shared address space is implemented directly in hardware, and only aggravates the communication bandwidth problem.

Layout techniques for NUMA shared-memory machines are a younger field than general-purpose optimization. In 2001, Jie Tao et al. developed a method for analyzing NUMA layouts in simulation [74] and found that applications that are not tailored for the layout of the specific NUMA machine frequently face severe performance penalties [73]. This expanded upon and confirmed work in 1995 by Chapin, et al. who analyzed UNIX performance on NUMA systems [14]. Abdelrahman, et al., among others, have developed compiler-based techniques for arranging arrays in memory to improve performance on NUMA machines [2], however the techniques are static and in addition to working only for arrays of data or other language-supported data types, they also optimize for specific memory layouts or a specific machine size.

1.5 Approach

Operating in an environment that supports lightweight, mobile threads and fast synchronization primitives in addition to shared memory opens many possibilities in terms of what can be done at runtime. To learn what operations or activities provide the greatest benefit, an analysis of the possible runtime layout modifications must be performed.

The first requirement of a good analysis is a good set of information about the way that large scientific codes tend to use memory. Rather than rely on small kernels or simplistic models, better accuracy is obtained by using memory patterns from large applications of the sort that would be used in a supercomputer setting.

Hardware that supports such features as lightweight threads and fast synchronization primitives, such as PIM, does not exist in a sufficiently large scale to study. In such situations, the most common approach is to simulate such a system, and that is the approach proposed here.

1.6 Outline

This paper discusses the basic design and basic components of a scalable multithreaded runtime for a massively parallel PIM-based computer, and in particular the options for memory management. Chapter 2 outlines the environment provided by a PIM-based system, and discusses some of the relevant challenges and approaches involved in even the simplest of runtime systems. Chapter 3 considers the potential for data migration and the consequences it would have on the architecture of the system, both in terms of overhead and opportunities and requirements for hardware support. Chapter 4 discusses the utility of designing basic system software to take advantage of the parallelism inherent in a PIM system,

and demonstrates the point by comparing the scalability over several numbers of processors of a standard memory allocation library with that of a parallel memory allocation library in simulation. Chapter 5 details the proposed extension to that preliminary work. Finally, Chapter 6 summarizes the thesis of this proposal.

CHAPTER 2

JUSTIFICATION

2.1 Features of a Basic PIM System

A processor-in-memory (PIM) system is an unusual architecture that addresses many of the performance bottlenecks in modern computers by grouping processing units and memory units onto the same chip. These systems must provide the necessary hardware support for the large-scale fine-grained parallelism that is needed to take full advantage of the inherent parallel design of the PIM architecture. While the details may vary between PIM implementations, there are two features which are fundamentally necessary. These features are lightweight threads and fast synchronization.

2.1.1 Lightweight Threads

The basic structure of a PIM unit is a simple lightweight processing core (or a set of cores) with fast on-chip memory. The connection between the core and the memory is short, fast, and wide.

That the core is lightweight means that it has a very simple pipeline; no reordering, no branch prediction, or any of the other standard conveniences of modern heavyweight processors, particularly those that require substantial area on the die. In such a lightweight core, data hazards and other sources of pipeline stalls

must be avoided through the use of multiple threads, which must be lightweight to expose sufficient parallelism. The core keeps a list of available, runnable, independent threads and interleaves execution of their instructions.

2.1.2 Synchronization

In any sufficiently parallel system, synchronization primitives are significant sources of delay and overhead. Though a system of very large numbers of nodes can be efficient when performing computations that are partitioned so as to eliminate data dependency between nodes, for most scientific applications this is unrealistic; such systems need a way to manage data dependency. Data dependency in shared memory computing is commonly managed through the use of locks and atomic actions. The number of locks in use by a given system or the amount of contention for the locks, is correlated to the number of nodes in the system, so locking overhead increases with the size of the system. Execution time, therefore, benefits greatly when locking is given consideration as a first-class memory operation supported by the hardware. As such, PIM units must have lightweight synchronization primitives built into the architecture, implemented as a standard memory operation.

An example of a simple locking primitive is a semaphore, which could easily be accomplished with either an atomic bit-flip, or the more standard test-and-set. Such a semaphore bit could, for example, be attached to memory words as a status bit that would make all operations on that memory word stall until the bit is cleared.

One of the more interesting lightweight synchronization techniques that relies on such a primitive is a fine-grained technique known as Full/Empty Bits (FEBs).

In a system that uses FEBs, every segment of memory (possibly every word) is associated with an extra semaphore bit, much in the way a clean/dirty bit is associated with pages in a page table. Read and write commands are then added to the architecture with semantics that respond to the state of that bit. For example, when data is written to a chunk of memory that chunk can be designated as "full" by setting the bit, and when data is read out the chunk can be designated "empty" by clearing the bit. It is an easy extension, then, to have a read operation stall if the chunk of memory it is directed to read is "empty" until some other thread marks the chunk as "full". Similarly, a write could stall when attempting to write to a chunk of memory marked as "full". Of course, it would be useful to have memory operations that do not respect the FEBs, as well as ways of manipulating the FEBs without modifying the associated chunk of data.

FEBs can be used to emulate other synchronization primitives, such as semaphores or mutexes, but are particularly well-suited for constructing "FEB pipelines", or circular buffers used for communication between a producer and a consumer that allow the either the consumer or the producer to transparently block until the buffer is ready for their operation. There is, of course, a trade-off to be examined between the granularity of the locked regions and the silicon overhead of implementation, but that is not the focus of this paper.

2.2 Functions of a Runtime

Operating systems, particularly in the supercomputing field, are viewed with a kind of disdainful appreciation; they provide useful and attractive conveniences to the programmer, but those conveniences come with a cost in terms of overhead. The goal of a supercomputer is to provide the maximum possible computational

power, and the conveniences of an operating system are often either too resource-intensive or too unpredictable to be tolerated [8]. While this often makes running without an operating system seem attractive, most systems do need some minimal ability to manage software and recover from errors.

A runtime, however, need not be the heavyweight feature-rich environment that is so often associated with the term "operating system". Instead, like the Parix system [39], it can be as simple as a set of libraries of common management functions that allow software to coordinate operations with itself more easily.

In a massively multithreaded system, runtime assistance is more easily cast as a combination of a set of simple threads performing useful functions and a collection of standardized data structures for doing things like memory management and message passing.

The purpose of a runtime in a supercomputer setting is similar to that of a library: to answer certain basic questions and perform certain basic operations so that they can be implemented reliably and can perform predictably, giving application programmers more time to do more useful work. These basic questions include:

- What is faster: data close together or spread apart?
- Is some memory local?
- How should memory (small fast chunks? large slow chunks?) be allocated and/or distributed?
- How should threads be spawned and distributed?
- Can there be too many threads, and under what conditions?

- Are there protection domains, and what are they?
- How do threads and processes communicate, particularly between protection domains?
- Can threads avoid overwriting each other accidentally?
- What happens when a memory reference fails?
- What happens when a thread crashes?
- What happens when a thread exits?
- How does a node avoid and recover from overload?
- Can a process avoid and recover from deadlock easily?
- Can a blocking operation time out?
- Can threads be debugged?
- Can a mobile thread or memory object be referred to persistently?

This is not meant to be an exhaustive list, but merely to give an idea of the utility in having standard answers to basic questions. Some of these questions may have simple or even obvious answers, however having answers is still useful.

2.3 Challenges

In attempting to answer some of the important issues involved in providing runtime support for applications, some obvious challenges must be addressed.

2.3.1 Memory Management

When hardware is designed to do things like buffer communication, it generally is designed with a static block of dedicated memory used exclusively for that task, such as on a network interface card or an L2 cache. This makes the memory management easy, but introduces the concept of failure and memory-full conditions that must be dealt with and preferably avoided. In a PIM-type system, the communication hardware's memory may not necessarily be separate from main memory. There are a few ways of handling this: the hardware can "request" memory from the memory management system when it needs more through the use of an interrupt or exception, or the hardware can simply be assigned a range of preallocated memory for such a purpose.

Preallocating memory is an easy and efficient method because it is similar to using a static block of dedicated memory. It is tempting to think that such memory need not be addressable by software, but such separation introduces unnecessary overhead since any data in such memory must be copied to an addressable location in order to be used. Additionally, as with any preallocated fixed-size buffer, the size may be non-optimal for the given computational load or communication pattern. A given computation may require a great deal of communication between nodes—particularly if the communication pattern is irregular and may have large spikes—which could overload a smaller buffer. Alternatively, a given computation may require very little concurrent communication and a large preallocated buffer may waste space that could instead be used for application data storage.

If the existing communication buffer is exhausted by the communication pattern, the hardware has two alternatives: refuse further communication or increase the size of the buffer. Refusing communication may have consequences beyond a

few lost memory requests. Memory requests may be retried, though this increases the communication overhead. If the communication buffer is used for outbound communication as well as inbound, this may prevent communication from being set up, which would require possible retries at multiple levels.

Dynamically allocating communication buffer memory may provide a solution to the drawbacks of a statically allocated communication buffer, but requires communication between the hardware and the application (or runtime) that is managing memory—which may not be local. Such communication may happen in many ways. For example, the hardware could understand the memory map format used by the runtime and could use it to locate and reserve unused memory. Such behavior requires a lot of hardware involvement in memory management and limits the creativity and flexibility of future memory management designs. It may be useful instead to have a special interrupt, hardware exception, or assigned thread to run in order to reserve additional memory for the hardware's communication buffer. This interrupt can be equivalent to triggering the memory manager's standard memory allocation function, though in that case the memory returned may not be contiguous with the already allocated hardware memory, which is likely an important detail for the hardware.

2.3.2 Memory Layout and Protection

The way in which memory addresses are arranged, both in the physical sense and in the logical sense, has significant consequences for memory management. There are many ways that memory could be arranged. A useful memory layout for a large shared address space system needs to provide two things: memory latency must be predictable, and memory must be readily accessed/addressed from any

node in the system.

2.3.2.1 Simple Direct Addressing

The simplest sort of memory map is to organize memory as a long array of sequentially addressed words, such as portrayed in Figure 2.1.

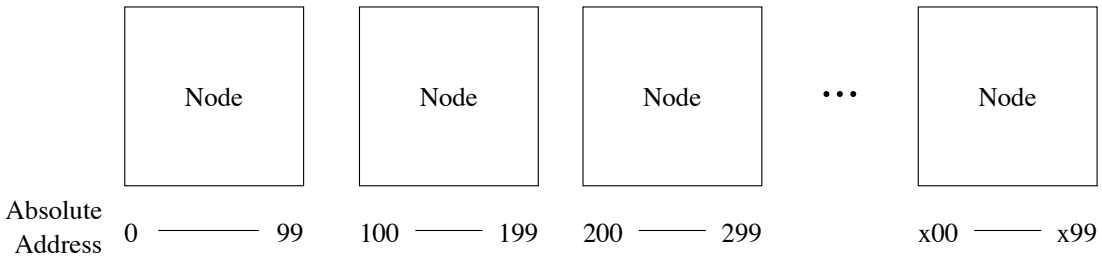

Figure 2.1. Simple Memory Layout

Memory that is addressed in this way is convenient for many reasons. Memory latency is quite predictable, because no address translation needs to be done and thus the location and latency of a given address is predetermined. Memory is also easily addressed from anywhere because no matter where in the system a memory reference is made, a given address maps to the same spot in the system.

Such a simplistic map has several drawbacks. Some of the primary drawbacks are the problems of persistent references and memory fragmentation. First, because all memory addresses always refer to the same location, memory references to mobile objects (such as threads) quickly become stale. It becomes difficult

to maintain a reference to a thread, or a thread's thread-local memory, that has migrated from node to node. It is possible for the program to maintain something like a thread table that is updated whenever a thread decides to migrate to a new processor, but this becomes a much larger undertaking in situations where common memory objects (data) can be migrated or if it is possible that objects can be migrated without the program's explicit command. In either case, a central repository of pointer indirections becomes a bottleneck.

The problem of memory fragmentation is more severe, for several reasons. Memory must be managed—marked as either in-use or available. If it is managed in a centralized fashion, the memory map becomes a bottleneck for all memory allocation in the entire system. If memory is managed in a distributed fashion, the partial memory maps distributed throughout the system fragment memory, and the precise location of each partial memory map defines the maximum size of memory that may be allocated as a contiguous block by a program. Even if such fragmentation is deemed acceptable, memory will be further fragmented by application use, which further constrains the maximum contiguous memory allocation possible at any given time. Virtual memory, in the sense of having a TLB, addresses the fragmentation problem for a single node. However, this merely transforms the fragmentation problem into a distributed data structures problem, as each node needs a way to reliably transform a virtual address into a real address. This is discussed further in Section 2.3.2.5.

2.3.2.2 Memory Partitioning

A modification to simple absolute memory addressing that is somewhat more useful is to segment memory. An example of this approach is illustrated in Fig-

ure 2.2.

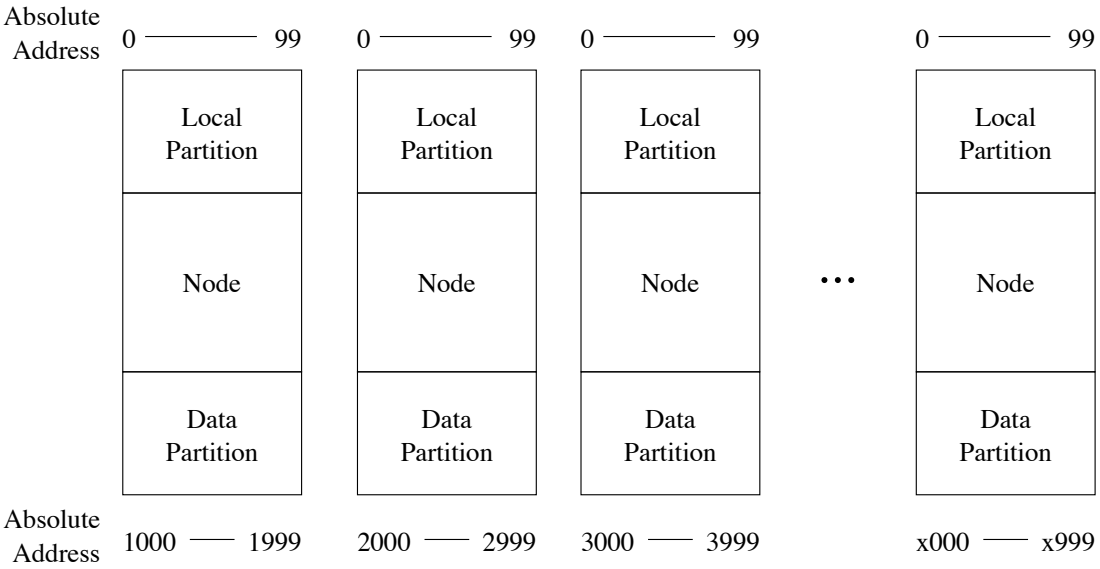

Figure 2.2. Memory Partitioning Example

Using a memory map like this, large data segments can be allocated easily. Local memory is then used for things like thread frames, memory management, and communication buffers. This memory map has most of the same problems as the previous method, such as memory fragmentation and persistent object addresses, but now also has the problem of lack of flexibility. If the same system is to be used for data-intensive applications where a great deal of data memory is needed but very few of threads are used and also for parallelism-intensive applications where very little data memory is used but a great number of thread

contexts are required, the system partition would have to be reconfigured before every application to tailor it to the profile of the application to be run.

2.3.2.3 Local Memory Aliasing

Another option in terms of memory maps is a simple local map, as illustrated in Figure 2.3. This is essentially a variant on base-relative addressing, where the base is location-dependent, and is always the first absolute address of the node the request is made from.

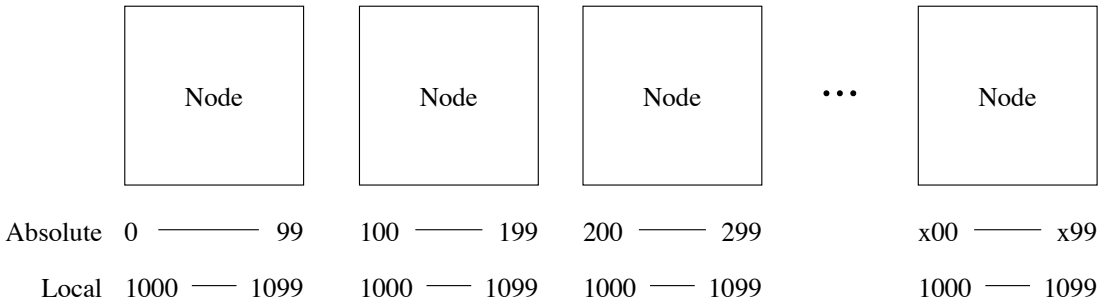

Figure 2.3. Local Memory Map

This allows node-specific addresses to work regardless of which node the request is made on. For example, libraries for communication may be loaded in the memory images of every node in the same location. This mapping conveniently allows a limited form of invisible thread migration. If a node becomes overloaded, it may be possible for a thread to be paused, for all of its thread-local memory and context to be migrated to another processor, and then restarted, without re-

quiring the thread to be modified or otherwise notified that it is now executing on a different processor. All of the node-local "scratch" memory that the thread may have allocated is precisely where it is expected to be.

This sort of memory aliasing is extremely cheap to implement quickly, while still providing some limited ability for overloaded nodes to transparently recover from overload. There is a significant restriction on this overload recovery method, however, and that is that the node-local "scratch" memory addresses used by a thread on one node must be unused on the node that it will be migrated to.

Rather than using a simple memory alias as just described, it may be preferable to use a more full-featured virtual memory address translation table for local-only memory. With such a system, transparent migration may be achieved by simply creating appropriate virtual address table entries on the new node for the migrated thread.

2.3.2.4 Thread-local Descriptor Tables

A common feature of desktop systems using Intel architectures is something called Local Descriptor Tables [31]. Each thread in the system could have, as part of its context, a pointer to an address map or "descriptor table." This descriptor table may be shared among several threads, or each thread may have its own. The descriptor table may also contain a chaining pointer, such that a thread may reference addresses mapped by its parent's descriptor table while maintaining its own independent address maps if necessary. This arrangement is very flexible, and provides a mechanism for dealing with memory fragmentation and for providing persistent addresses to mobile objects. The drawback of this method, however, is speed. Global addresses defined in the first parent's descriptor table may create

a bottleneck if kept there. However, copying them from the parent's descriptor table into each child's descriptor table creates a significant coherency problem with associated overhead. Additionally, the maximum time for an address lookup may be unbounded as the memory system tracks down each successive, chained descriptor table.

2.3.2.5 Globally Persistent Addressing

To deal with memory fragmentation, and provide persistent object addresses, a global address map may be used. Like a virtual addressing system, this global address map maintains mappings between persistent virtual addresses and absolute addresses. The absolute-addressed memory space may be broken into pages, each of which can be allocated separately, and mapped by this persistent distributed global address map.

With a persistent distributed global address map, several operations become possible that were not before. Large ranges of memory may be easily allocated that would not fit into the memory belonging to a single node. Memory fragmentation would no longer restrict allocation size either, and the question of locating a thread or other object in the system becomes merely a matter of knowing its persistent address and having the global mapping updated whenever threads or memory pages are migrated.

The precise details of such an address map is both critical to performance and implementation specific, however it could be done with a distributed hash table [72, 81]. Another, more simplistic method, may be to use a simple broadcast query for all allocations and lookups, though this would dramatically increase the communication bandwidth requirements.

Such a powerful address mapping allows both the application, the hardware, and the runtime to move blocks of memory or threads for any reason, transparently. Such behavior could be done to cluster memory near a thread that accesses it, or to move a thread closer to the memory it is accessing without losing any links the memory or thread may have already established with other memory objects. A distributed memory map with persistent addresses may also be useful for surviving situations where local memory on a given node is in short supply: less frequently used memory blocks can be migrated to less congested nodes to free up local memory without interrupting execution.

The primary drawback of a distributed arbitrary address map is speed: both lookup and allocation times are likely to depend on the size of the overall system. While this may scale reasonably in the sense that lookup times can be a logarithmic function of the size of the system, this would still severely slow down memory accesses. It may be quite possible to make local access of persistent addresses much faster than remote accesses, but this is still much slower than using absolute addresses or more simplistic aliasing and remapping techniques. Such overhead may not need to affect all memory references. While it may be useful or even required to have such a system for locating shared, globally accessible data reliably in a massively parallel system, many threads are likely to want local "scratch" memory for temporary calculations that does not need to be shared or globally accessible.

2.3.2.6 Thread-specific Virtual Memory

Another interesting idea for memory allocation and remapping would be to associate each application thread with a memory thread (or threads). Memory

references to a set range of addresses could be handed to that thread's associated memory thread in order to be fulfilled rather than be fulfilled directly. This provides for a great deal of flexibility: threads that need to share references to large address blocks can share memory-threads or can share memory data structures, while threads that don't need to share references like that do not need to. One might even be able to construct sets of shared memory mappings that can be defined at runtime, in a manner similar to MPI "communicators", particularly since in many cases only the size and contiguousness of the memory blocks, but not the exact addresses, must be maintained between application threads. This technique could even be used to implement a globally shared address space.

The execution model could be further refined to farm out all memory operations to independent threads. In other words, each computational thread could depend on several memory threads whose job is to make sure that the computational thread has the right data in the right registers (or thread-local scratch space) when it is needed.

2.3.3 Exceptions

There are many forms of exceptions that may need to be handled in even a minimal-feature-set system, including hardware faults, program faults, interrupts, and system calls. There are, similarly, many ways that a programmer may wish these exceptions to be handled, including being handled by another process-defined thread, being reported to another thread via a queue, ignoring them, or even forwarding them the parent runtime, among other options.

One common type of exception that the runtime often needs to be able to handle is an interrupt, such as a timer interrupt or a communication interrupt.

There are several ways that a runtime could handle such exceptions. A common method in modern processors is an interrupt vector that specifies the address of a string of instructions to execute in the event of each variety of exception. This could also, in a PIM system, spawn a thread, however an interrupt vector is impractical in a massively parallel system for several reasons.

It is inefficient to copy code to every node in the system for handling all possible kinds of exceptions, most of which will never be executed. The ability to keep only the exceptions that must be handled locally local is extremely useful. Unfortunately, communication itself (and thus, the act of forwarding exceptions) may be a source of exceptions, and such exceptions must either be ignored or handled locally.

In many cases, using an interrupt vector to handle exceptions also presumes a large amount of atomicity in the exception handling that can be very undesirable in a massively multi-threaded environment. Many operating systems find it necessary to ignore all new interrupts while any interrupt is being handled. Some, like Linux, break interrupts into an uninterruptible top half and an interruptible bottom half. In a massively multi-threaded environment, such policies would likely lose too many exceptions to be practical.

A more efficient technique in a massively multi-threaded environment with hardware-supported locks and threads may be to queue exceptions in a special exception queue that is processed by an independent exception thread (or threads) that handles each exception in turn. This technique for exception-handling would have a very small footprint and conceptually allows a great deal of configurability in exception-handling. For example, another queue may be used to allow processes or threads that wish to be notified of specific types of exceptions within specific

scopes to register to be spawned or notified when such an exception happens. New kinds of exceptions may be defined at runtime. At the same time, handling for unexpected or undefined exceptions may be forwarded to a central system exception handler if and when they happen without adding significant complexity to the exception thread or the hardware's exception handling.

2.3.4 Multiwait

A single thread waiting for a memory address state, such as a read waiting for the memory address to become full, is relatively easy. When the requested memory address is marked as "empty", for example, the address of the thread that requested the address is written to the contents of the memory and a flag is set to indicate that the empty memory contains a thread address. Then, when another thread writes to that address such that it would be marked "full", the value is sent to the thread indicated in the memory address.

Once a second thread may be waiting for that same memory address, the implementation becomes much more complex. There are many ways that have been proposed for handling more than one thread attempting to access the same locked (or "empty") word in memory.

2.3.4.1 Polling Wait and Multiwait

The most trivial method of waiting for a memory state is to poll the memory location in question (i.e. a spinlock). This method does not suffer from the problems inherent in interrupt-based locking and waiting (discussed later in this section) because there is no need to keep track of what threads are waiting for a given address. On the other hand, polling has the downside of overhead that

increases with the length of time a thread is required to wait. The overhead associated with a simple poll operation increases linearly, though this can be improved by using more complex exponential back-off strategies. On systems with heavyweight threads which may need to do complicated process swapping, the overhead of polling is relatively minuscule, though it may cause large numbers of process swaps. In a system where threads are lightweight, process swapping takes very little effort, and where more complex waits and locks are relatively simple (i.e. supported by hardware), polling has a relatively high overhead and becomes a practical alternative to interrupt-driven waiting only for extremely short expected wait times. The exact length of time polling may be used before it becomes more expensive than an interrupt-driven wait obviously depends on the implementation of the poll and the overhead of context switching (if any).

2.3.4.2 Thread-based Multiwait

One technique for handling multiple waiting threads is to spawn a new thread to multiplex the wait. In other words, when a thread requests a memory address that is, for example, marked "empty", if the address is also marked as already having another thread waiting for it, a new thread is spawned that will replace the original blocking thread. This new thread will block on the target memory address twice, sequentially, once to write the value back to the current requesting thread and again to write the value back to the original requesting thread (or to replace the original requesting thread's position as the only waiter). The original thread's address is replaced in the requested memory address by this newly spawned thread's address. The basic implementation of this concept, as demonstrated in Figure 2.4, is relatively straightforward. The order of dereferencing,

however, is important to the sanity of the process.

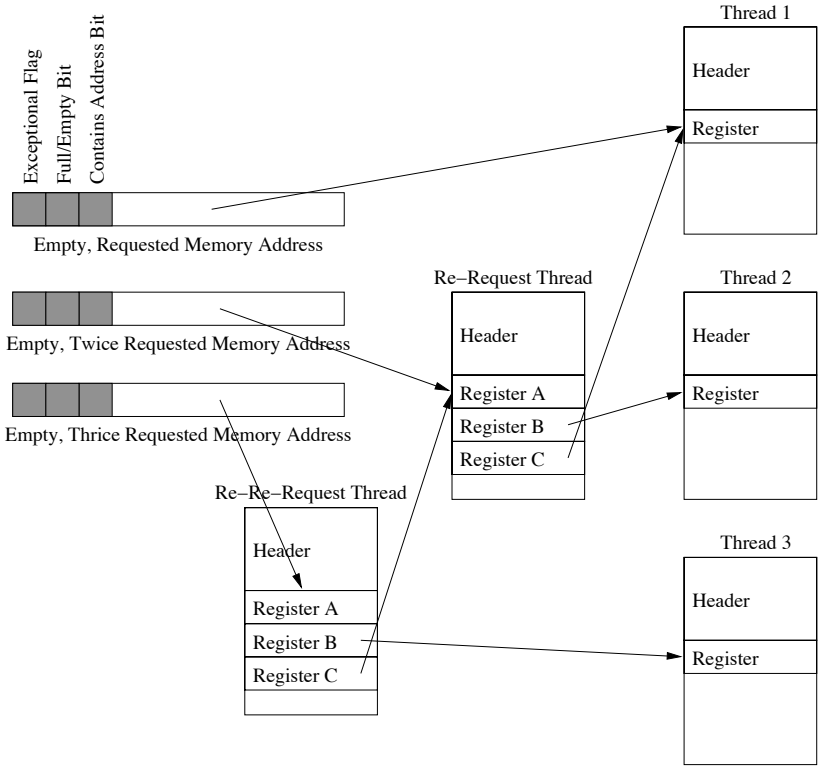

Figure 2.4. Thread-based Multiwait Technique

There are many ways to accomplish this kind of multiplexing, including a hardware implementation. Regardless of the implementation details, however, it is critical that replacing the currently blocking thread be an atomic action. Without direct hardware support, the easiest way for a processor to do this is to, for a short period of time after the new wait thread is created, while it is being installed, ensure that only one thread can access this memory (possibly by

ensuring that only one thread can execute). When the wait-thread is created, add the current thread's address, as defined by the read call, to the thread's contents. Then, ensure that only one thread can access this memory. Once only the current thread can access the memory, re-try the memory access in case the state changed while the wait-thread was allocated. If the memory is still being blocked by another thread, read out the address of that thread and store it in the wait-thread. Finally, write the wait-thread's address into the requested memory address and re-enable universal thread access to the memory address. This sequence of events safely minimizes the amount of time the hardware spends in a single-thread-only state.

2.3.5 Queue-based Multiwait

The memory overhead of thread-based multiwait is significant and may be an issue for large numbers of waiting threads. Additionally, there is an implicit FILO ordering to memory requests using that technique which may cause some memory requests to starve or may be otherwise undesirable. It may be desirable to have priority in memory access, for example. A similar technique, using a queue, is more powerful and may be more useful. In this case, when multiple threads are waiting on the same memory address, they are added to a queue. Rather than waiting directly on the memory address, the threads are actually waiting on an shadow address in the queue that will be filled by a single wait thread. This technique is demonstrated in Figure 2.5.

In this case, there is a speed trade-off on when to create the thread with the queue: on the first access or on the second. In general, if there are expected to be multiple requesters, it may be useful to set up the wait-queue with the

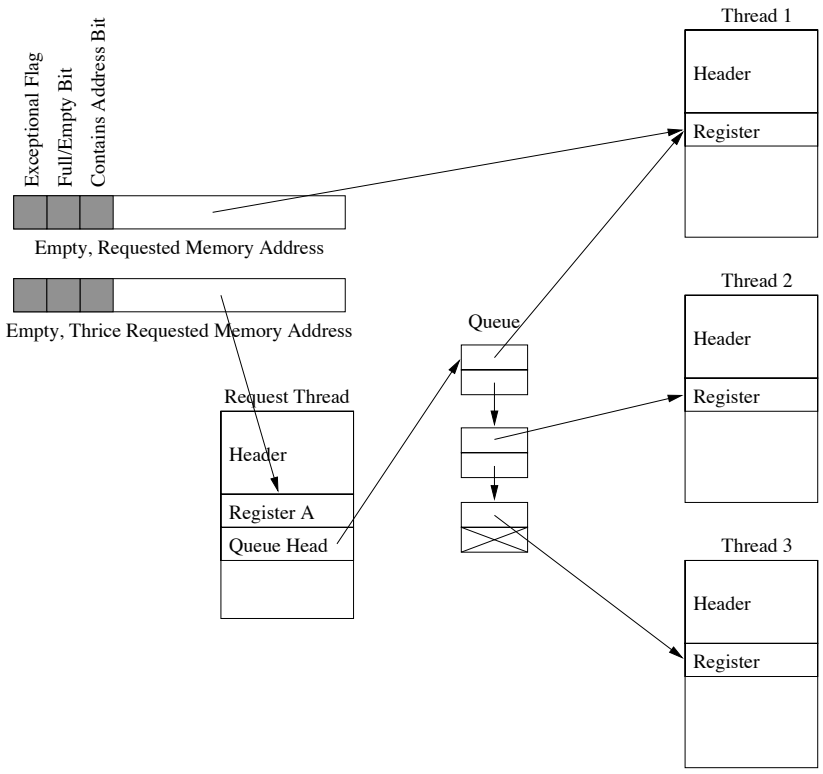

Figure 2.5. Queue-based Multiwait Technique

first memory request. This is, however, not necessary until the second memory request. Waiting for the second memory request, however, requires many of the same machinations as the thread-queue multiwait technique.

Of course, because using a queue requires locks, there is the possibility that locks within the queue may require safe multiwait. This circular dependency may be handled in several ways. The basic method would be to combine queue-based multiwait with another multiwait strategy: use queue-based multiwait for the general case and another one that does not rely on safe queues for locks inside a queue-based multiwait's queue. Either thread-based multiwait or polling-based multiwait can be safely implemented.

2.3.6 Safe Deallocation and Reuse

It may be desirable for the runtime to clean up after a thread or process that has been terminated, regardless of the reason. It may also be desirable for the runtime to sanitize freed memory for reuse, such that it may guarantee that there are no outstanding memory references to that freed memory. These two operations are related. One step to cleaning up a thread or process would be to deallocate any memory specific to the thread or process. It is, however, entirely possible that such memory contains addresses that are tied up in unfinished memory operations; in other words, the memory has "dependencies". For example, another thread or process elsewhere in the larger system may be blocked waiting for one of the freed addresses to be unlocked or filled with data (using FEBs). As another example, the memory may have previously belonged to a thread context structure and parts of it may be awaiting the result of incomplete memory commands that were issued by the thread. If there are such outstanding memory commands, unless these commands can be prevented from affecting memory after they have been issued (for example, with some sort of protection scheme or virtual addressing scheme), the memory cannot be reused safely, as its contents may be corrupted by the outstanding memory commands. How this is handled depends on exactly what sort of dependencies affect the freed memory.

If some of the freed memory addresses are awaiting the result of memory commands (like a load), there are two basic methods for handling the dependency: by canceling the unfinished memory requests, or by simply waiting until the outstanding memory requests finish. If, on the other hand, the freed memory addresses are being waited on by other threads, there is really only one thing to do: return an error, akin to a "communication failed" error, to the blocked threads. These two

categories of memory dependency can be referred to as the "unfinished command" category and the "remote waiter" category, respectively.

2.3.6.1 Detecting Dependencies

Dealing with memory dependencies requires, first and foremost, detecting that such dependencies exist. To a large extent, this is implementation dependent. For example, a convenient hardware implementation could add a bit flag to every address indicating whether it was the subject of a memory dependency or not, thus allowing memory to be safely deallocated in $O(n)$ time, where n is the size of the memory being deallocated. Without such a flag, the software would need to record the same information in some other way, either through a special library or by intercepting all loads, stores, and lock requests and recording them somewhere that the memory allocation library can find them.

In some special cases, some assumptions may make this task easier. For example, if the freed memory is known to have been a thread header, and if thread headers can be protected from other threads placing arbitrary, possibly accidental, locks on any address within the thread header, and assuming that the thread's code may not be modified at runtime, then the algorithm for releasing that memory can be made simpler. In such a case, only a very few specific locations in the thread header may be dependent on outstanding memory requests, and whether they depend on unfinished memory commands might be determined by examining the state of the thread before it exited.

It is possible, of course, to simply ignore this problem and simply blame any memory corruption on poor programming. As this is likely frequently desirable in very high-performance code, it would be necessary to be able to turn such safety

features off. Nevertheless, such precautions against unintentional memory corruption should at least be possible, preferably with as little overhead as possible. Doing this by using an interface like the exception queue is unfeasible because of the speed and frequency that monitored memory requests would have to be enqueued in the exception thread's queue. A hardware extension is the fastest method.

Assuming that memory is labeled with the flags described in Section 2.3.4, a simple rule may be used to detect memory dependencies. If loads and stores use the "contains address" bit in the address header to indicate incomplete or complete memory requests (without modifying the "Exceptional" flag), then most simple memory dependencies may be detected by simply examining the "contains address" flag. The exceptions to this are memory dependencies that must not set the "contains address" flag such as block-if-full-write, which are relatively easy special cases and if they are implemented as part of a library will simply reduce to the other cases. Polling dependencies, however, may be impossible to detect.

2.3.6.2 Waiting

The simplest way to handle "unfinished command" dependencies is to wait until they are finished before allowing the affected memory to be reused by the system. Assuming that there is a simple way to do this, when memory is freed it must be checked for unfinished memory commands. If this memory was originally a thread header, the thread may be removed from the queue of active threads, because it has exited, but until its memory is dependency-free, it must be periodically re-checked for unfinished memory commands.

It is relatively simple to establish a persistent "Death Thread" that has an

associated safe queue of memory blocks. When memory is freed and determined to contain unfinished memory commands, it can be added to the Death Thread's queue. The Death Thread then periodically (either continuously or based on a sleep function) iterates through it's queue rechecking each block of memory. Once the memory commands any given block was waiting for have completed, the Death Thread can dequeue the related memory and add it back to the pool of free memory. This, of course, has the drawback that memory requests may, for whatever reason, not complete in a reasonable time frame. In extreme cases, for example in deadlock, this may cause memory starvation.

2.3.6.3 Canceling Memory Requests

A more powerful method for dealing with outstanding memory requests, that can handle both kinds of memory dependencies, is to cancel them. This, of course, depends somewhat on the exact mechanism that remote memory requests use for waiting. If the hardware supports it, this can be a relatively simple matter. If the memory dependency is a non-blocking unfinished command, cancellation is probably excessive and may not be particularly useful: unless something is significantly delaying inter-PIM messages, waiting is likely a better approach in this case as long as non-blocking unfinished commands can be distinguished. If the memory dependency is the result of a blocking memory command, in either the "unfinished command" or "remote waiter" categories, it may be (and possibly must be) canceled.

Memory requests can be divided into cancelable and uncancelable categories. It is possible to—either with the compiler, an explicit library call for all memory accesses, or with some method of intercepting memory requests—ensure that the

only uncancelable memory requests are either local to the node or are performed by a persistent thread (i.e. part of the runtime). A cancelable request, presuming the hardware does not support cancelable memory operations natively, is a meta-request composed of three uncancelable requests between parts of the runtime. For example, if a thread makes a possibly blocking request for a remote memory address, the request could be re-routed to a local memory "shadow" of the remote address. A runtime thread could then communicate this request to the remote node which would establish the memory request on the remote node. When the action completes on the remote side, it would communicate the result back to the local runtime thread which would in turn update the memory shadow which would then be fed to the local thread as if it came from the remote node. If the local thread exited before the remote request completed, the runtime could send a cancellation message to the remote node. The remote node would dequeue the request in the same way it enqueued the request, and the local runtime would set the shadow memory address's header bits to allow the local thread's memory request to complete quickly. The dependency structure of such a request is illustrated in Figure 2.6.

A similar mechanism could be applied to all memory references regardless of the location of the target, thereby allowing not only for blocked memory requests to be cancelled, but also to debug memory dependencies if desired. This mechanism, of course, would introduce a great deal of overhead into the system, but would make memory references a great deal more robust and would prevent unfinished memory commands from holding memory hostage.

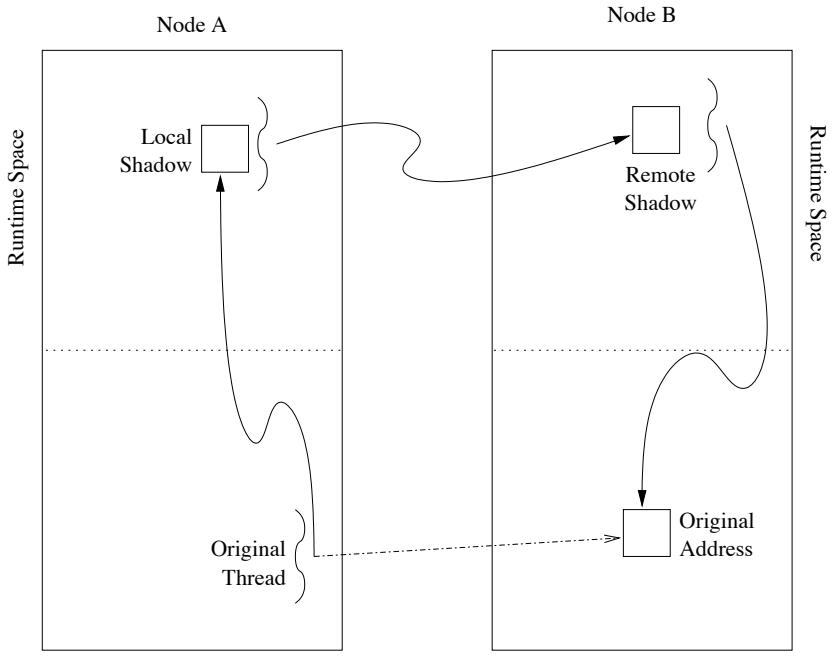

Figure 2.6. Cancelable Memory Operation

CHAPTER 3

MIGRATION

3.1 Reasons to Migrate

The easiest form of memory allocation is, of course, static allocation—memory is given a location when it is allocated and that location does not change. While this is simple and makes finding objects in memory relatively straightforward, it limits the amount of adaptivity that can be employed. In commodity computers, this behavior is worked around in many ways; caching systems move memory between RAM and CPU-specific caches and paging systems in the OS move memory between RAM and disk storage. In both cases, while the address referring to a particular object in memory does not change, the location that address refers to changes rather frequently in response to how that piece of memory is used.

3.1.1 Congestion and Overload

The typical reason that memory objects in commodity processors are migrated from one storage medium to another is in response to overload. When too much has been stored in the CPU's cache, some of it is evicted and sent to a less quickly accessible place in the memory hierarchy (usually, the RAM) to make room for more recently used memory objects. And when too much is being stored in RAM, some of it is evicted and sent to a less quickly accessible place in the memory

hierarchy (the disk) to make room for more recently used memory objects. While a PIM-based system might not readily move storage to disk, it will most likely still run into overload conditions. Indeed, a PIM must consider more overload conditions than simply too many things to store in memory. In particular, the overload may also be that there are too many threads on a single PIM—in other words, the PIM is too congested. Even if the PIM is not technically overloaded, performance may be improved by employing overload-recovery techniques to load-balance threads to other PIM nodes.

3.1.2 Proximity to Computation

In classic NUMA systems, the placement of memory objects must account for more than simply whether there is sufficient space to store things. The proximity of that storage to the location where the stored data is being used must also be taken into account. Similarly, in a PIM context, memory residing on a given PIM node is most quickly accessible from that PIM. Requesting memory from other nodes is significantly slower. Thus, a thread incurs a latency penalty when it uses memory that is on a different PIM node than that on which the thread is executing.

3.1.3 Predicted Costs

The situation of a thread using distant ("remote") memory is one which is a particularly important decision point for PIM computation. To reduce the latency penalty, there are two possible options: either move the thread to be closer to the memory, or move the memory to be closer to the thread.

A naïve way of looking at the problem is to simply compare the relative costs

of transfer and the cost of ignoring the penalty. If the remote memory is small and will only be used once, it is obviously easiest to leave it everything where it is and simply pay the latency penalty. If the cost of transferring the remote memory to the thread's node, both in terms of bandwidth and in terms of time, is greater than the cost of copying the thread to the remote memory's node, then the thread should be sent to the memory it wishes to use. If the reverse is true, and copying the thread is more expensive, then the memory should be transferred to the thread's node. While these are logical decisions, the evaluation of the costs is far more complex than the simple copy operation.

When transferring a block of memory from one place to another, it may be necessary to prevent access to that memory by other threads while the transfer takes place, in order to avoid memory corruption. Indeed, in addition to preventing access, the calculation of the cost of transferring the memory should theoretically include the cost the move will impose on other threads who also may be using that memory. For example, if one PIM node has a hundred threads all using a local block of memory and another thread on a distant PIM node also wishes to use that block, the cost of transferring that block to the remote node includes forcing all of the hundred threads already using the memory block to start making remote memory operations.

There are additional costs to consider for transferring a thread from one place to another as well. First, unless the hardware understands relocatable threads, any outstanding memory operations must be resolved before a thread can be migrated, just as when a thread exits (see Section 2.3.6). Secondly, the cost of transferring a thread includes not only the cost of transferring the thread's execution state, but also references that it will make in the near-future. For example, if a thread makes

a remote reference to one node and the next thousand memory references will be to a different node, sending it to the first node is probably unwise. Unfortunately, foreseeing future memory references is an intractable problem; the best solution is to assume that future references will be similar to previous references. It is worth noting that commodity CPU caches have the same problem and resort to the same approximation.

The thread may also have a certain amount of state tied up in thread-local "scratch" memory, which must be transferred along with the thread—using threads without such thread-attached memory removes this potential cost.

3.2 Implementation Considerations

When controlling the layout of memory at runtime, there are two basic methods: to layout memory as it is allocated, and to alter the layout of memory as the program progresses and the use of that memory changes. If sufficient controls are exposed to the application programmer, layout at allocation time can be a powerful mechanism, in part because it is so simple. With layout allocation, the choice of location for the memory need only be made once, which means that it has essentially the same overhead that naïve allocation does. However, the single decision point is also a limitation; if the use pattern of that memory changes over the course of the application's execution, its location may become a liability. Allocation-time layout can, at best, provide a good "average" location that is good some of the time and not excessively bad the rest of the time.

3.2.1 Addressing

In order to support runtime data relocation, or "migration", the system as a whole must provide portable addressing (also known as pointer indirection). The most common form of this is virtual addressing. Because of the overhead involved in providing global virtual addressing, virtual addresses are most commonly organized into pages: chunks of memory that are relocated as a whole. Thus, global virtual addressing ensures that any page in memory may be migrated at any time from anywhere in the system to anywhere else. While the ability to relocate memory easily is good, the fact that global virtual addressing relies on aggregating memory into relocatable blocks can be a significant limitation and inefficiency. When any memory in a page can be migrated, the entire page must be migrated.

An alternative to global virtual addressing is limited virtual addressing, where memory can be specified as relocatable when it is allocated. This might be implemented by reserving a particular address range for relocatable addresses. Addresses within this range can be requested with a special function call, and the simplistic range-based relocation allows the memory controller to quickly decide whether a given memory address is virtual or not. Limited virtual addressing can thus permit memory to be relocatable on a much finer-grained level without requiring the overhead that would be necessary to do global fine-grained virtual addressing. Even so, the overhead—both in terms of bookkeeping data structures and lookup time—for fine-grained addressing is significantly higher than for course-grained. The benefit, however, may be significant.

3.2.2 Speed

In addition to memory efficiency, one of the primary considerations in a relocatable data scheme is the speed of operation. The speed is a factor in three places: allocation, deallocation, and use. The "use" category is at once the most critical, as it is the common case, and the most complex, as it includes any alterations in access time due to data migration. When handling runtime access of relocatable memory, are two possible states that the memory may be in: static, and "in motion". The static state is fairly self-explanatory: when the memory is not being migrated, and the only overhead is the time it takes to look up the address translation. The speed of this lookup depends primarily on the way in which relocatable data is stored. If, for example, it is stored in a simple binary tree, the lookup can take as little as $O(log(n))$ time where n is the number of relocatable memory blocks. If the memory blocks are instead stored in a distributed hash table, the lookup may take as much as $O(log(N))$ time, where N is the number of nodes in the system. The "in motion" state is the obvious complement to the static state: the memory is currently being copied somewhere. How this is handled may depend on the amount of incoherency the system can tolerate, but preventing (or simply stalling) modifications during migration is a simple technique for preventing memory corruption. Such a policy, though, introduces the potential for significant delay while memory is migrated.

Allocation and deallocation speeds are, while not as critical, still extremely important. A common optimization to ordinary high-speed code—particularly in threaded code—is to implement memory pools, so as to avoid using `malloc()` and `free()` and thereby avoid incurring the overhead they require. In general, this makes sense, as the programmer can use much simpler memory tracking than

is needed to support `malloc()`'s generality, and can implement using different pools for each thread. In a system that uses passive runtime data relocation, it is important for the system to know when a block of memory is not needed anymore. Standard pooling techniques make unused memory indistinguishable from used memory, and so can result in unnecessary memory traffic.

CHAPTER 4

PRELIMINARY WORK

4.1 Demonstrating the Potential

A simple parallel memory allocation scheme was implemented on a PIM architectural simulator and compared with a naïve memory allocation scheme, to demonstrate both the problems with using an existing allocation scheme, and the benefits of even a trivial customized scheme.

4.1.1 The Simulator

The PIM architectural simulator used is the Structural Simulation Toolkit (SST), written and designed by Arun Rodriguez in cooperation with Sandia National Laboratories. The simulator is capable of simulating and executing large systems and large real-world programs in the form of PowerPC binaries, instruction trace files, and even MIPS binaries. The toolkit includes a set of standard libraries, including libc, libgcc, and several others based on the MacOS X library suite, so that executable binaries can be compiled for it using a standard and unmodified compiler. It simulates a system with two gigabytes of memory.

The libraries included with SST have been modified to support the basic functionality of the SST simulator. The pthreads library, for example, is a wrapper around the low-level lightweight threads possible in the SST's PIM architecture.

The locks in the libc library also use Full/Empty Bits rather than the standard test-and-set methods.

4.1.2 The Naïve Memory System

SST's original set of libraries was pulled directly from MacOS X. As written, they relied on many services provided by the operating system, such as Mach system calls and a virtual memory system. Rather than create a virtual memory system within SST, a small change to the library was made, replacing the system calls that request virtual memory pages with a simple lock-protected `sbrk()`-like function that maintains a very simple stack-like structure of memory. Because SST only simulates a single program at a time, this allocation scheme is sufficient to correctly execute even large codes and standard benchmarks.

Underneath this simplistic memory allocation technique, the memory map modeled by SST is somewhat unusual. When simulating an all-PIM system in SST, memory is associated with an array of homogeneous PIM chips. Each chip has one or more processing cores. Memory is divided among the processing cores in a round-robin fashion in blocks of 4,096 bytes (this number is configurable). This memory is also divided among the PIM chips in blocks of 16,384 bytes. Because of this memory layout it is impossible to allocate more than 4,096 contiguous bytes on the same processing core.

4.1.3 Malloc Modification

The memory allocation subsystem (`malloc()`) in the SST's libc was replaced with a new one designed to more fully exploit the hardware's parallelism. Instead of a stack-like structure, memory is logically divided into contiguous segments

associated with each processing core. For example, because SST simulates a system with two gigabytes of memory, if there are four processing cores each core is assigned by this new malloc a contiguous 512 megabyte segment of memory; if there are eight processing cores, each core is assigned a 256 megabyte segment.

Within each segment, memory is allocated in logical "blocks" of 4,096 bytes. A memory use map is reserved within each segment for storing the in-use state of each logical block within the segment. Meta-information structures, each one block big, are defined to make tracking and freeing memory easy. They are allocated as-needed. The structure of an example segment is illustrated in Figure 4.1.

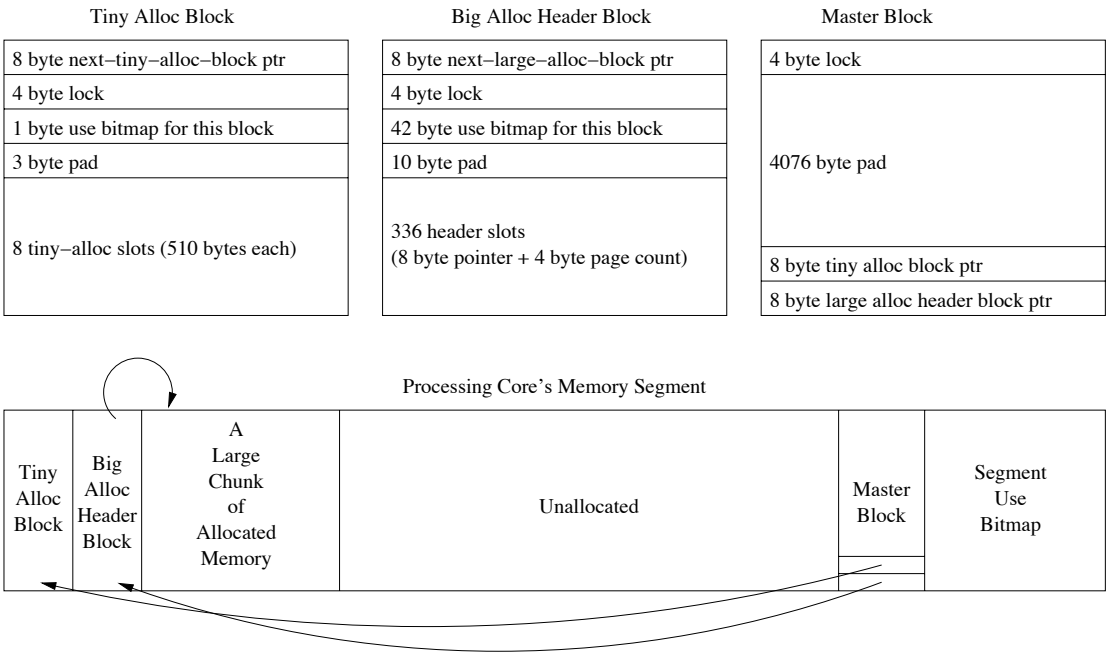

Figure 4.1. Simple Parallel Malloc Structure

This scheme for allocation of memory is simplistic and basic, but is very parallel. Every meta-information block has its own lock that must be obtained in order to read or write meta information from that block. The bitmap for the segment is protected by the lock in the master block. This allows, first and foremost, every processing core to allocate memory independently of the other processing cores in the system. Allocations of memory within a single segment can achieve some amount of parallelism as well, because each meta-information block can be manipulated independently.

Memory allocations are divided into two categories: tiny allocations and big allocations. Tiny allocations are allocations of 510 bytes or less. These are assigned slots within a tiny-alloc block. This arrangement makes tracking and freeing such small allocations rather efficient. Big allocations are anything larger than 510 bytes. These allocations are tracked in the big allocation header blocks as tuples consisting of a pointer to the beginning of the allocated memory and a counter of how many blocks belong to that allocation. Thus, when a pointer is given to the standard `free()` function, it can be quickly deallocated. If the pointer is aligned to a block size, it is a big allocation, and otherwise it is a tiny allocation. Big allocation pointers are looked up in the big allocation header blocks to discover their size so that the appropriate number of blocks can be freed from the memory segment's block-use bitmap. Tiny allocation pointers are marked as free in the tiny allocation block's slot-use bitmap.

4.1.4 The Comparison

To demonstrate the power of designing parallel system components, such as the memory allocation library, to improve the scalability of a system over a more

simplistic approach, a small benchmark was created. This benchmark has a single parent thread that uses the pthreads library to spawn 1,024 child threads, and then waits for all of the child threads to exit. Each of these child threads calls malloc to allocate first a small amount of memory (four bytes) and then a large amount of memory (4097 bytes). Then each thread frees its memory.

As each thread is spawned, the pthreads library allocates local memory for a pthread thread header and then allocates memory for the new thread's stack. The thread's stack is allocated in memory that belongs to the processing core the thread will be spawned on. When the thread exits, the thread's stack is deallocated. When the parent thread has finished spawning threads, the pthread thread headers are cleared as each child thread is joined by the parent thread.

This benchmark was compiled and linked against both the original malloc implementation and the new malloc implementation, and was simulated in SST with varying numbers of processing elements. The execution time was determined to be the number of cycles executed by the first processing core because that is the core the parent thread executes on. The results are presented in Figure 4.2.

Because both implementations are so different (and because the original is so simplistic), the absolute number of instructions is hard to compare. Instead, the percentage performance improved for each number of processing cores versus the two-core case is demonstrated. The difference is dramatic. Not only is the parallel malloc more scalable, but it is also much more predictable. The naïeve malloc is able to benefit somewhat from more processors, but gets overwhelmed by 16 processors, and contention for the single malloc lock reduces the performance over the 8 processor system significantly. It is likely possible to improve the performance and scalability of the parallel malloc even further through tuning

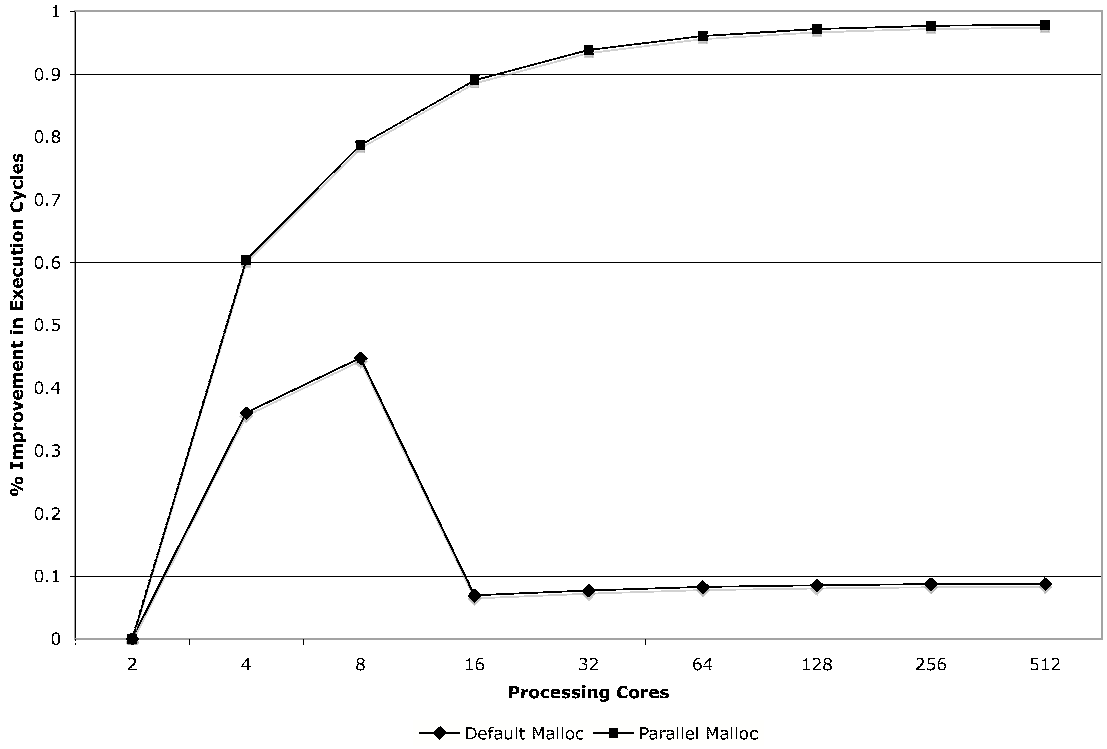

Figure 4.2. Scalability of 1024 Threads Calling Malloc

of the pthreads library to take additional advantage of the memory available on other cores by not allocating the pthread header locally, and by optimizing the malloc library for the size of memory allocations used by the benchmark or by breaking up the segment block-use bitmap and allowing different sections to be locked by separate locks.

4.2 Preparing for Further Research

For a further and more concrete examination of the benefits and trade-offs involved in large-scale multithreaded scalable runtime support for applications on PIM architectures, examining the specific performance implications of memory

management practices is both a powerful proof of concept and a useful tool for development and customization of large applications for the architecture. Efficient data layout is typically considered the responsibility of the programmer, and as such one thing a parallel memory management scheme must do is provide the programmer with the ability to layout data in a predictable and explicit way. However, it is also the case that important multithreaded applications are difficult to analyze, evaluate, and predict due to their complexity. Generic methods for both analyzing an application's memory behavior and predicting the effect of data layout on execution time would be extremely useful.

To that end, a more detailed examination of the memory behavior of real applications and the effect that the possible alterations in memory allocation and mapping may have on the memory performance of those applications will need to be performed.

4.2.1 Collecting Trace Data

The goal of collecting traces of the memory behavior of real applications is twofold. First, the information can be used to map the dependencies between explicitly parallel operations and thus get a better idea of the effect of host-based changes upon the performance of parallel applications as well as the opportunities for optimization through communication reordering. Second, as MPI operations in a shared memory machine are essentially memory reads and writes, a trace combining MPI operations and memory operations can be transformed into an access dependency graph, which can be used for analyzing the performance ramifications of modifications to data layout and distribution schemes of large MPI-based codes were they to be operated within a shared memory or PIM context. MPI is

not, however, necessary. Tracking the memory access patterns of standard serial applications may also be of use.

4.2.1.1 Reasonable Parallel Behavior

One of the key elements to collecting good trace data is choosing applications that will behave realistically. This is a special challenge when dealing with architectures that don't exist yet, such as PIM. Thankfully, there is an architecture that provides similar parallelism tools, such as fast locks, full/empty bits, and lightweight threads. The architecture, called the Multi-Threaded Architecture (MTA) [1], has some large-scale software available for it. In particular, the Multi-Threaded Graph Library (MTGL) [7] provides a good example of high-performance code written for an architecture that provides these capabilities.

The platform-specific features of the MTA are primarily used through the use of C-language pragmas that instruct a custom MTA compiler as to how to parallelize the code. Because MTA applications are so dependent on the compiler, porting such code to other architectures rather difficult. To assist in making the MTGL portable to other architectures, a library was developed that provides the MTA's hardware features in a generic and platform-independent way. The library is called the qthread library, and includes a set of features known as the futurelib that simplifies the threading interface. An implementation of the library has been developed for both the SST PIM simulator and regular SMP systems.

Surprisingly, the qthread library makes parallelizing standard code extremely simple. As a demonstration of it's capabilities, the HPCCG benchmark written by Mike Heroux for Sandia National Labs was parallelized with the futurelib interface to the qthread library. The HPCCG program is a simple conjugate

gradient benchmarking code for a 3-D chimney domain, largely based on code in the Trilinos [26] solver package. With simple modifications to the code structure, the serial execution of HPCCG was transformed into multithreaded code. As illustrated in Figure 4.3, the parallelization is able to scale extremely well. Those performance numbers came from running the code on a 48-node SGI SMP.

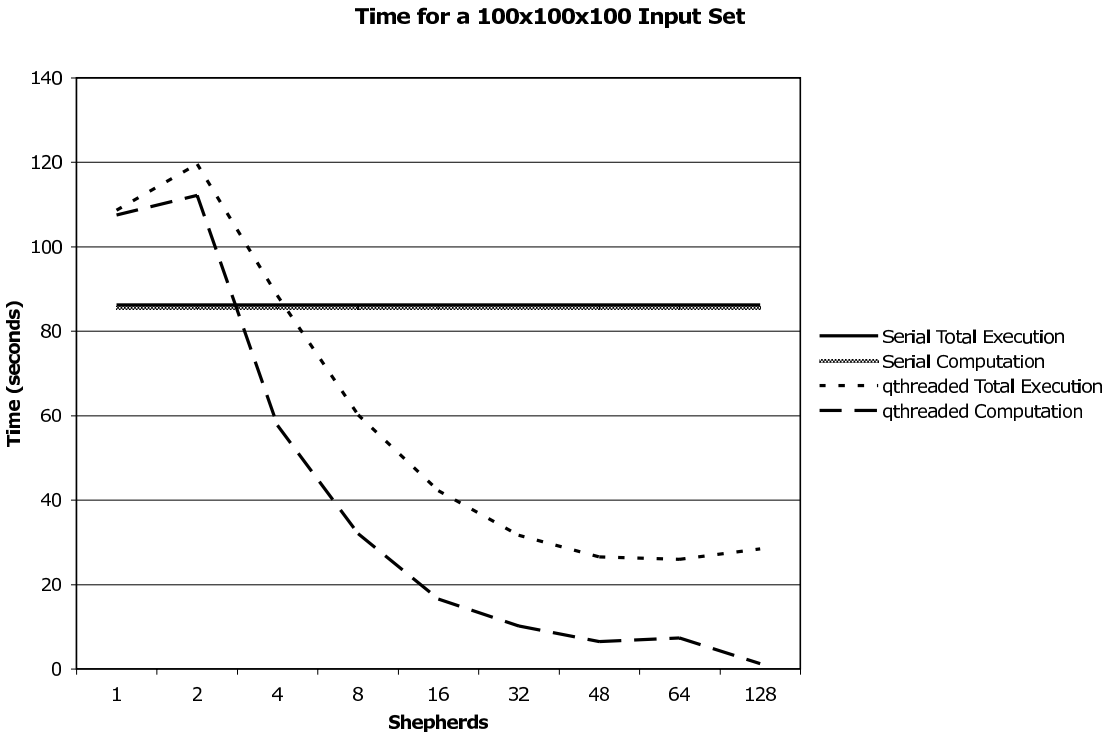

Figure 4.3. HPCCG Benchmark on a 48-node SMP

4.2.1.2 GDBFront

GDBFront is a pair of programs designed to track memory accesses in terms of programmer-visible objects. Specifically, in units the size of variables and allocated blocks. There has been previous work in analyzing memory use patterns by address, evaluating its temporal locality and spacial locality [46]. This adds to that previous work information about how the memory addresses are related in programmer-visible chunks that could be rearranged in memory at either compile-time or runtime to improve performance, but that must be rearranged as a block. While addresses commonly represent merely a single byte of data, the data is related by what it is used for. For example, a 4-byte integer represents 4 different memory locations (in byte-addressed systems) that, for speed, are usually arranged sequentially in memory and used as a chunk. However, if the four bytes are treated by the application as a character array or a bit field, each of the four bytes may be accessed with different frequencies or in different patterns. An analysis of memory access patterns that does not account for restrictions on the relocation of memory may naïvely recommend rearranging the component bytes of the 4-byte integer, or may waste resources tracking each byte independently.

A more interesting example would be matrix multiplication: in general, when multiplying two matrices and placing the results into a third, the standard compiler-defined memory layout is to group all memory from each two-dimensional array together. Based on the stride length, however, it may be beneficial to performance to interleave the elements of the arrays together. This tool records each variable's size and address, allowing for such analysis.

GDBFront analysis of a program is a two-step process. The first step is to use the Gnu Debugger (GDB) to discover all of the programmer-visible information

about the program's runtime memory use, including names, sizes, and offsets. This information is dumped to a file, which is read in by the second program in the two-step process: refTrack. This program uses the Amber instruction tracing program from Apple Computer [5] to collect an instruction trace of the program being evaluated. Memory accesses can be extracted from the instruction trace and the affected addresses are mapped back to a variable name and context, which is then logged in a trace file. Dynamic allocations can also be determined at runtime by monitoring the program counter and when it reaches the address of the beginning of a `malloc()` or `free()` or related call, extract the arguments to the function from the register state.

4.2.1.3 Trampi

The initial motivation for the development of trampi was to fill the gap in MPI tracing libraries, which is that current MPI traces are very static. They record MPI operations as merely operations that occur at a specific time relative to the program invocation. This allows researchers some ability to determine how altering the characteristics of the communication network would affect the performance of the MPI communication, but does not allow for estimates of the affect altering the characteristics of the hosts would have on the performance of the traced application.

To improve this situation, trampi uses the Performance Application Programming Interface (PAPI) library [11] to obtain information about the program's behavior between and within MPI calls. The information that can be collected is limited by what PAPI can collect on the given hardware platform, but generally it can collect things like the cycle count, cache miss rate, and instruction mix,

among other things. This data is recorded for every processing block between MPI calls, as well as for within each MPI call.

4.2.2 Performance Modelling

Once trace data has been collected, it can be used to examine the effects of data layout on performance by examining each memory operation and determining the amount of time each will take based on given memory layouts and given policies for runtime data layout. Using a discrete-event simulator such as the SST (Section 4.1.1), factors such as thread locality, bus contention, memory latency, and even memory access dependency relationships can all be used to determine the delay in memory access times and its effect on the memory access pattern, which can then be used to extrapolate the effect of data layout on execution time. A full-system simulation can also be used to more fully model the precise details of execution, to give a much more accurate picture of the expected effect of data layout on performance.

CHAPTER 5

PROPOSED WORK

5.1 The Question

When examining and attempting to predict memory behavior in a large-scale multithreaded system, there are many questions to answer. A basic question to answer is: how much of an effect does data layout have? Certainly it has a significant one, but how significant? One way to answer this question is to compare the performance of some benchmark software using naïeve layout to the performance of the same software with an optimal layout, however this approach immediately begs the question of the definition of optimal. Just as cache strategies are compared by looking at their relative number of cache misses, a good metric for comparing layouts is to compare the number of off-chip memory operations. The perfect data layout would succeed in preventing all off-chip memory references and is most likely impossible for all but toy programs. However, such a presumption does give a useful (if unreasonable) base for comparison.

A question along the same lines that must be addressed is how much—if at all—system-level interference can improve data layout, and thus performance. Certainly the programmer can lay out data in convenient and fast ways, but can the system library functions, such as `malloc()` and virtual addressing schemes, conspire to provide significant performance improvement? If just those functions

in their standard form cannot provide significant performance gains, perhaps alternative methods or altered semantics can. For example, creative byte-level virtual memory remapping can not only move and distribute stack and static memory around the system, but may be able to distribute large allocated chunks of memory in a way that improves performance sufficient to justify the complexity. Perhaps a companion, predictive pre-fetcher thread may be useful for ensuring that data is where it needs to be. However, as no predictive pre-fetcher operating without the programmer's explicit instructions can be perfect, it is possible that the bandwidth consumed by moving data unnecessarily will negate the benefit.

A possible feature of a large multi-threaded system is the ability to migrate an execution thread from one place to another, to chase the data it needs to operate upon. Deciding whether migrating the thread or migrating the data will provide the best performance requires more analysis than simply asking whether transferring the data or the thread state takes more bandwidth: the congestion of the nodes, future accesses, thread communication, and topology all must be taken into consideration, however the relative impact of these factors on general-case performance is unknown.

Even presuming that data layout is statically defined at the beginning of the program, one must still determine whether clustering a given set of data is a better policy than distributing it, so that intelligent decisions about the static layout can be made. Some indicator characteristic to guide programmer layout of given data structures and kinds of memory is something that would be exceedingly useful. The effect of changes to data layout are difficult to predict accurately due to the complexity of the applications that are intended to run in such multithreaded environments. Generic methods for both analyzing an application's memory be-

havior and predicting the effect of data layout on execution time are extremely useful, even if not used for runtime layout modifications.

The proposed dissertation will provide an analysis of several data layout policy options, to provide an idea not only of the benefits intelligent runtime support could have, but also of the potential for performance improvement data layout analysis offers in a PIM-style distributed system.

5.2 Instrumented Memory Operations

Memory operation traces will be collected from several significant codes to examine and compare their memory behavior. There are several extremely large computationally-intensive parallel applications available from Sandia National Laboratories that can be used for this purpose, such as LAMMPS [58, 59, 60], Salsa [66], sPPM [54], CTH [27], and the MTGL. Additional traces will also be collected from more common applications such as `vim`, `ls`, and `gcc` that will make an interesting comparison to the larger scientific codes.

The applications were chosen for several reasons. As the goal is to predict performance of applications in an extremely large-scale parallel environment, applications that have been designed to handle variable amounts of parallelism are likely to be the type of applications that are made to run on such systems. Applications operating in a PIM-like environment are likely to need to be able to handle arbitrarily large amounts of parallel execution. The architecture and behavior of a program that is designed to take advantage of arbitrary levels of parallelism is more likely to be representative of the behaviors of programs designed for a PIM-like system. The chosen parallel programs—LAMMPS, Salsa, sPPM, and CTH—have all been designed to take advantage of as much parallelism as is available. The

MTGL is a multi-threaded graph library designed for the MTA architecture [1]. It uses a very large number of threads and is as such can also take advantage of arbitrary amounts of parallelism. Because its parallelism is based in threads rather than in MPI, it more closely represents the type of code that would likely be run in a production-scale PIM system. The other three programs—`vim`, `ls`, and `gcc`—were chosen for comparison purposes. The intent is to demonstrate the differences in memory behavior, and how an adaptive runtime can improve the performance of both. It is possible to extract a surprising amount of parallelism from serial code [63], which makes even these programs potential candidates for use on a PIM-like architecture.

The traces collected will include a temporal record of all memory operations—loads, stores, allocations, and deallocations—organized by time, program counter, and the affected relocatable data blocks in the application. Currently, the Trampi (Section 4.2.1.3) library records MPI operations, the address and size of the buffers and structures involved, and counts instructions, cycles, and cache misses in between each operation. In a shared memory system, it is likely that MPI communication will more closely approximate passing around a pointer to a buffer rather than actually copying the data (in benchmark applications like sPPM, more than 14MB is transferred in most MPI operations). The GDBFront tool currently records traces of the variables that are being operated upon by the traced program. The data from these two tools can be combined, producing a comprehensive record of the memory operations that the traced programs make in a multithreaded context.

Information about the relocatable data blocks is necessary to model potential data redistribution policies. The state that must be communicated in a thread-

migration must be considered more than merely the registers of the thread, but also it's current and near-future working set of memory. With a comprehensive record of memory operations, the "working set" of the program can be quantitatively measured and used to estimate how much state the program currently needs at each MPI operation, and what memory it will use before it does the next operation.

The MPI use-pattern of scientific applications tends to be cyclic: communicate, compute, communicate, compute. This is illustrated well by Figure 5.1, which was obtained from the Trampi library while tracing a benchmark run of the LAMMPS program running on eight nodes of a development cluster (RedSquall) similar to the RedStorm supercomputer. The vertical red stripes consist of the raw IPC measurements, which occur at MPI boundaries; in essence, each vertical stripe indicates a burst of communication. At each communication burst, the working set of each thread changes, allowing for the possibility that a thread may migrate and take less state with it than would typically be necessary at other times during execution.

The record of memory operations can then be used to determine the size of the state that would needs to be transferred. With this information, a simulator could track not only the effect of layout policy on data layout, but the effect of data layout and redistribution policies on the memory access latencies experienced by the application, including total latency time, average latency, and latency jitter.

In a PIM-style system (Section 2.1), everything involves memory access. As such, data layout and distribution have a greater effect on performance than on virtually any previously explored architecture. The proposed dissertation will explore several facets of data layout and distribution to determine the behavior

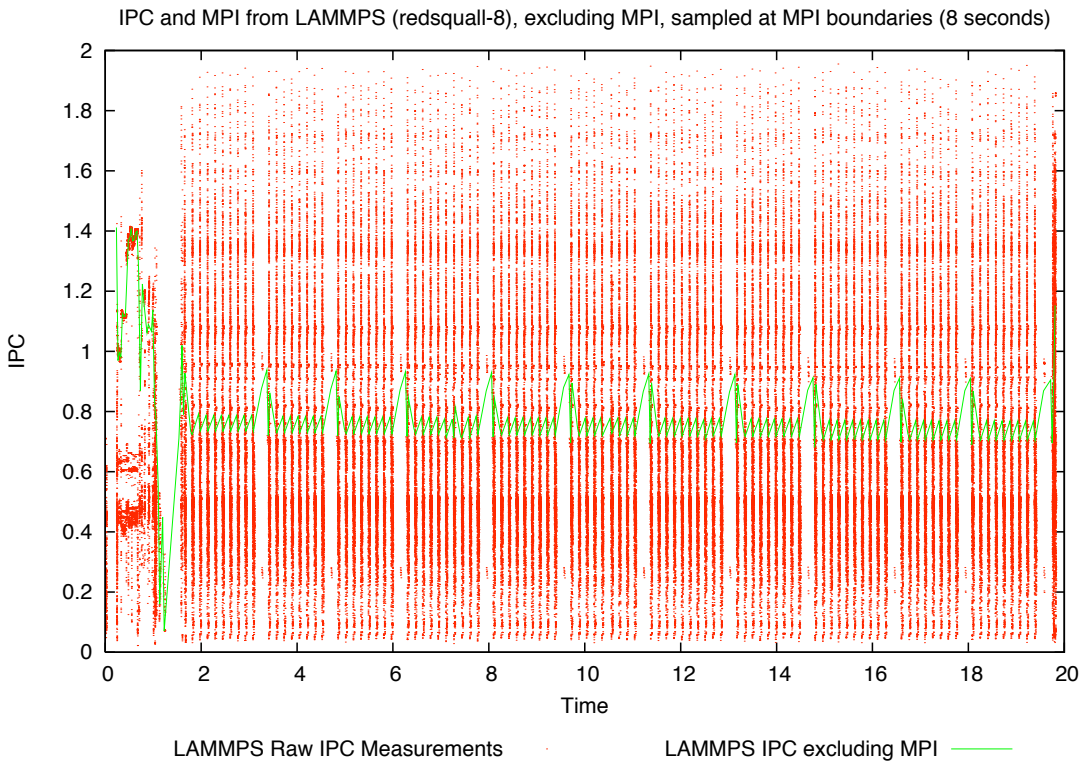

Figure 5.1. IPC and MPI from LAMMPS, collected on RedSquall using eight nodes

exhibited by a PIM-like system in simulation. PIM-style systems are sufficiently dynamic, and the intended type of program is sufficiently complex, that compile-time data layout is unlikely to be optimal without significant effort on both the compiler's part and the programmer's part. While a component of the proposed research is to provide means of analyzing data layout to predict performance, part of it is to determine good methods of making layout decisions at runtime.

5.3 Simulation

There are three basic categories of data blocks in a program: global, scoped, and allocated. Each of these may be treated by different policies or similar policies, and the benefits of each must be examined. For comparison, however, it is necessary to establish a baseline, based on current standard and relatively simple allocation methods. The most basic policy that closely approximates existing systems will use a naïve stack-based local-node-only data distribution scheme for persistent stack variables, with no data duplication or distribution. Global variables will be loaded into memory at a standard location on the base node, much like current single-system behavior, and will not be copied to other nodes. The memory allocation scheme will be a simple sequential addressing scheme, similar to standard BSD-style memory allocators. Additional policies will build off of this base.

A similar policy that instead distributes the global variables among the nodes may provide beneficial load-balancing, and so may reduce the average latency at the expense of jitter. Global variables may instead be copied to all nodes, requiring a coherency protocol between them. This policy will answer the question of whether the overhead of coherency is less expensive than remote data accesses when sharing global data. The same may be attempted for runtime-allocated memory, however this may require assuming extremely large per-node memory banks. If there is a benefit to maintaining copies of global data, this may also be compared to maintaining copies of all application data on all nodes. While this policy is unlikely to be feasible or desirable in a real system due to the extreme memory requirements, it will provide an idea of the asymptotic performance were the application programmer to treat more and more data as global.

Another category of modification that may be made to the baseline policy is to alter the memory allocation algorithm. As a basic improvement, modelling a parallel allocation scheme that makes each node's allocation independent of the others must be examined (this may be compared to the results reported in Section 4.1.4). It also may be beneficial to attempt to load-balance memory allocation around the system, or to load-balance among neighboring lower-latency nodes, and so these modifications will be examined as well.

The third category of modification that may be made to the baseline policy concerns stack variables, or scoped variables. The most obvious behavior is to treat all stack variables as private and exclusively local. However, once MPI communication of the data stored in these variables is taken into account, it may be more useful to treat some of them as globals that need to be copied to multiple nodes. If the system is presumed to use thread-migration, where the traced program may relocate, the question of what to do with these scoped variables takes on greater importance: should they be copied, transferred, or left in their original location? This question is somewhat entwined with the question of when threads should migrate to a different node. In a real PIM system, it would likely be possible to explicitly define migration behavior, but for simulation purposes, a heuristic must be used instead. A simple behavior, for example, is to have each thread migrate whenever it reads from or writes to a non-local memory address. Migration, however, has a cost associated with it, both in terms of bandwidth and latency. That cost depends on the amount of state that must be transferred. Thus, it may sometimes be beneficial to migrate the thread, but only if the cost of migration is lower than the cost of doing the remote memory accesses in the access pattern, which of course depends on how much memory (i.e.

scoped variables) must be migrated with the thread. An alternative would be to distribute scoped variables among the nodes or among nearby nodes.

In essence, each of these policies must be examined. Along with the policies, the effect of different thresholds, based on the amount of state that must be transferred and the amount of data that needs to be accessed, defining when to migrate a thread should be examined.

5.4 Schedule

The proposed thesis has three primary phases: trace collection, simulation, and analysis.

Trace Collection The first task of the first phase is to integrate the two existing memory-reference and MPI tracing tools, to collect what amounts to a threaded trace of memory references and performance distances between them. This is expected to take approximately three to four weeks. Once the tools have been integrated, the sample applications must be built and run, and trace data must be collected and collated. This process should take no more than an additional month (many representative benchmark inputs take no more than an hour to complete). Which benchmark inputs to use is generally straightforward, and good benchmark input sets have already been collected.

Simulation The next phase of research will be to prepare the simulation environments. There are two simulation environments that will need to be prepared: a basic memory latency modeler, and a more complete execution model. Both environments will be based on the SST simulator (Section 4.1.1). The basic memory latency modeler will be a simple discrete event simulator that takes the trace data

as input, uses the SST memory model of a PIM-like system to gauge latencies, and estimates both execution time and total memory latency for the input trace of memory references. This is expected to take three to four months. Performing the simulations in all the necessary combinations will likely take at most two months. Then, two representative policies from the simple discrete event simulator (a simple policy and the best-performing complex layout policy) will be implemented as part of the `libc` that comes with the SST simulator. This is expected to take approximately two months. The SST will also need to be instrumented to record memory latencies. This is expected to take two to three weeks. Once these implementations are stable, two of the sampled applications will be compiled for the SST simulator, and run several times with both memory allocation schemes to confirm the results of the previous simulations, which should take an additional week.

Analysis The second phase complete, the final analysis phase may begin. This will include a paper-and-pencil analysis of one of the simpler programs (either sPPM or `ls`), explaining the behavior observed while under the several policies. This analysis should take approximately a month. Finally, writing the thesis may begin. This is expected to take approximately six to nine months.

5.5 Hypothesis

The expected result of the proposed research is an indication that the optimal data layout and distribution policy depends on the specific application. This would indicate that while generic runtime layout decisions may offer some benefit over all applications, significantly better performance is to be gained by specify-

ing the layout in the application. The performance of global variable policies is likely to vary depending on how actively used the global data is: if it is infrequently modified, the overhead of keeping multiple copies coherent will be greatly reduced. Allocated memory performance is likely to show a significant benefit from even simplistic layout heuristics, such as keeping small allocations local and load-balancing large allocations between nearby nodes. For example, in sPPM, buffers over 14MB in size are allocated for use in MPI only to be communicated and deallocated, where the receiving side then allocates a similar sized buffer for the receiving side to use. It would be more efficient to simply allocate the buffer on the remote node, copy the data there, and then inform the remote process of the location of this buffer. This expected improvement from heap allocation is also likely to depend on the migration scheme in some applications. For example, the MTGL uses much more easily migrated threads, as they are designed to have very little associated state. The performance of scoped variable accesses is likely to depend even more heavily on the migration policy, but is expected to show that local placement of scoped variables is usually optimal, unless they are either used for communication or are always used with a specific data set that does not move.

It is possible, however, that a single combination of these options provides significant benefits over the naïve scheme for all of the applications tested and that while customization may provide additional improvements, a single generic scheme may provide surprisingly good performance relative to the optimal performance.

5.6 Significance

The performance benefits discovered by the proposed research will not only demonstrate the utility of runtime services, but will also provide important infor-

mation on the relative benefits of data layout for application performance. The research will also provide tools for analysis of the memory behaviors of generic applications. This data can be used to estimate the performance benefits arbitrary applications may experience in a PIM context. It may also be useful for defining categories of memory behaviors which programs fall into, which would be of use to application programmers and designers who attempt to tailor their application to a PIM-like machine, even without runtime services to assist in data layout.

The exploration of the trade-off in copying memory with migrating threads will also provide valuable information about the benefits of migrating threads. In particular, it should demonstrate the effectiveness of partial-state migration. The research will either demonstrate the feasibility of a migrating thread with a partial stack and local state that is not register-bound, or will demonstrate that the overhead of even a partial stack is too great for realistic migrating threads. Either answer has significant implications for PIM-related research, particularly for compiler work and other simulation work that currently makes assumptions about the requirements of migrating threads or the overhead involved in migration.

CHAPTER 6

CONCLUSION

This paper proposes an architecture for a scalable runtime for a massively parallel computer based on embedded processor-in-memory technology. System software can be designed in a lightweight, dynamically scalable way that both has the capacity to provide advanced operating system features and can be trimmed down to a minimum-overhead, minimum feature-set for maximum computational power. The design of such system software is strongly dependent on the inherent parallelism and overall architecture of the hardware, and this paper focuses on a PIM-style architecture. The general features of such an architecture, as well as the inherent challenges to traditional runtime and operating system designs, are laid out in Chapter 2. An example of dynamically scalable lightweight operating system component design, organized in layers, is described in Chapter X. The utility of designing system software that is appropriate to the unique features of the hardware is demonstrated by the example memory allocation library that is tested in Chapter 4. This library is compared to a standard memory allocation implementation, from the MacOS X libc, that does not account for the parallelism or use the lightweight locking mechanisms provided by the hardware. The comparison demonstrates dramatic improvements in the scalability of the memory allocation when the structure of the allocation data structures is appropriate for the hardware's design. The basic proposal, detailed in Chapter 5, is to continue

to explore the benefits of memory allocation as well as memory relocation in such highly parallel, large-scale shared memory architectures. The layout of data, and the possibilities for runtime rearranging of data, will be evaluated in the context of a PIM-like system where data relocation must be balanced not only against the cost of relocation, but also against the possibility of thread relocation. The proposed dissertation will provide a means of evaluating possible data layout schemes, and the opportunities for runtime policies to affect performance.

BIBLIOGRAPHY

1. Cray MTA-2 system - HPC technology initiatives, November 2006. URL `http://www.cray.com/products/programs/mta_2/`.

2. Tarek S. Abdelrahman and Thomas N. Wong. Compiler support for array distribution on NUMA shared memory multiprocessors. *Journal of Supercomputing*, 12(4):349–371, 1998. ISSN 0920-8542. doi: http://dx.doi.org/10.1023/A:1008035807599.

3. Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, J. Kubiatowicz, B. H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *ISCA '98: 25 years of the international symposia on Computer Architecture (selected papers)*, pages 509–520, New York, NY, USA, 1998. ACM Press. ISBN 1-58113-058-9. doi: http://doi.acm.org/10.1145/285930.286009.

4. Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera system. *Tera Computer Company*, 1999.

5. *Amber Trace Format Specification v1.5*. Apple Computer Inc., 2006. Available as part of Apple Computer's CHUD tool suite.

6. J. L. Baer. 2k papers on caches by y2k: Do we need more? Keynote address at the 6th International Symposium on High-Performance Computer Architecture, January 2000.

7. Jonathan W. Berry, Bruce A. Hendrickson, Simon Kahan, and Petr Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Proceedings of the International Parallel & Distributed Processing Symposium*. IEEE, 2007.

8. Ron Brightwell, Rolf Riesen, Keith Underwood, Trammel Hudson, Patrick Bridges, and Arthur B. Maccabe. A performance comparison of Linux and a lightweight kernel. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster2003)*, December 2003.

9. Jay Brockman, Peter Kogge, Shyamkumar Thoziyoor, and Edward Kang. PIM lite: On the road towrds relentless multi-threading in massively parallel systems. Technical Report TR-03-01, Computer Science and Engineering Department, University of Notre Dame, 384 Fitzpatrick Hall, Notre Dame IN 46545, February 2003.

10. Jay B. Brockman, Shyamkumar Thoziyoor, Shannon K. Kuntz, and Peter M. Kogge. A low cost, multithreaded processing-in-memory system. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 16–22, New York, NY, USA, 2004. ACM Press. ISBN 1-59593-040-X. doi: http://doi.acm.org/10.1145/1054943.1054946.

11. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 42, Washington, DC, USA, November 2000. IEEE Computer Society. ISBN 0-7803-9802-5.

12. P. Budnik and D. J. Kuck. The organization and use of parallel memories. *IEEE Transactions on Computers*, C-20:1566–1569, December 1971.

13. Arthur W. Burks, Herman H. Goldstine, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument (1946). pages 39–48, 1989.

14. John Chapin, A. Herrod, Mendel Rosenblum, and Anoop Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 1–13, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-695-6. doi: http://doi.acm.org/10.1145/223587.223588.

15. III D. T. Harper and J. R. Jump. Vector access performance in parallel memories using skewed storage scheme. *IEEE Transactions on Computers*, 36(12):1440–1449, 1987. ISSN 0018-9340.

16. Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, 1970. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/356571.356573.

17. Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–241, 1999. URL `http://citeseer.ist.psu.edu/ding99improving.html`.

18. T. H. Dunigan. *Denelcor HEP Multiprocessor Simulator*, June 1986.

19. S. J. Eggers and R. H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 257–270, New York, NY, USA, 1989. ACM Press. ISBN 0-89791-300-0. doi: http://doi.acm.org/10.1145/70082.68206.

20. Domenico Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17(11):614–620, 1974. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/361179.361195.

21. Robert Fitzgerald and Richard F. Rashid. The integration of virtual memory management and interprocess communication in accent. *ACM Transactions on Computer Systems*, 4(2):147–177, 1986. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/214419.214422.

22. Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994. URL `http://citeseer.ist.psu.edu/17566.html`.

23. Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, William Athas, Vincent Freeh, Jaewook Shin, and Joonseok Park. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 57, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-091-0. doi: http://doi.acm.org/10.1145/331532.331589.

24. Karim Harzallah and Kenneth C. Sevcik. Predicting application behavior in large scale shared-memory multiprocessors. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 53, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-816-9. doi: http://doi.acm.org/10.1145/224170.224356.

25. John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 1996. ISBN 1-55860-329-8.

26. M. Heroux, R. Bartlett, V.H.R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, et al. An overview of trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

27. E. S. Hertel Jr., R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A

software family for multi-dimensional shock physics analysis. In R. Brun and L. D. Dumitrescu, editors, *Proceedings of the 19th International Symposium on Shock Waves*, volume 1, pages 377–382, Marseille, France, July 1993.

28. W. Daniel Hillis and Lewis W. Tucker. The CM-5 connection machine: a scalable supercomputer. *Communications of the ACM*, 36(11):31–40, 1993. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/163359.163361.

29. M. Holliday and M. Stumm. Performance evaluation of hierarchical ring-based shared memory multiprocessors. *IEEE Transactions on Computers*, 43 (1):52–67, 1994. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/12.250609.

30. Chris Holt, Jaswinder Pal Singh, and John Hennessy. Application and architectural bottlenecks in large scale distributed shared memory machines. In *ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture*, pages 134–145, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-786-3. doi: http://doi.acm.org/10.1145/232973.232988.

31. *IA-32 Intel Architecture Developer's Manual*. Intel Corporation, 2002.

32. Dongming Jiang, Brian O'Kelley, Xiang Yu, Sanjeev Kumar, Angelos Bilas, and Jaswinder Pal Singh. Application scaling under shared virtual memory on a cluster of SMPs. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 165–174, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-164-X. doi: http://doi.acm.org/10.1145/305138.305190.

33. Yi Kang, Michael Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *Proceedings of the 1999 IEEE International Conference on Computer Design*, pages 192–201, Washington, DC, USA, October 1999. IEEE Computer Society. ISBN 0-7695-0406-X. URL `http://citeseer.ist.psu.edu/kang99flexram.html`.

34. Suzanne M. Kelly and Ron Brightwell. Software architecture of the light weight kernel, catamount. Technical report, Sandia National Laboratories.

35. T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *Readings in computer architecture*, pages 405–417, 2000.

36. Peter Kogge. The EXECUBE approach to massively parallel processing. In *Proceedings of the 1994 International Conference on Parallel Processing*, August 1994.

37. Peter Kogge, S. Bass, Jay Brockman, Daniel Chen, and E. Sha. Pursuing a petaflop: Point designs for 100 TF computers using PIM technologies. In *Proceedings of the 1996 Frontiers of Massively Parallel Computation Symposium*, 1996.

38. Janusz S. Kowalik, editor. *On Parallel MIMD computation: HEP supercomputer and its applications*, Cambridge, MA, USA, 1985. Massachusetts Institute of Technology. ISBN 0-262-11101-2.

39. Falk Langhammer. Scalable operating systems, or what do a million processors mean? In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, page 310, New York, NY, USA, 1993. ACM Press. ISBN 0-89791-599-2. doi: http://doi.acm.org/10.1145/165231.166109.

40. D. H. Lawrie. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24:1145–1155, December 1975.

41. Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *PODC '86: Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 229–239, New York, NY, USA, 1986. ACM Press. ISBN 0-89791-198-9. doi: http://doi.acm.org/10.1145/10590.10610.

42. Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: a modular reconfigurable architecture. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 161–171, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-232-8. doi: http://doi.acm.org/10.1145/339647.339673.

43. Sally A. McKee. Reflections on the memory wall. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, page 162, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-741-9. doi: http://doi.acm.org/10.1145/977091.977115.

44. Gavin Michael and Andrew Chien. Future multicomputers: beyond minimalist multiprocessors? *SIGARCH Computer Architecture News*, 20(5):6–12, 1992. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/141408.141410.

45. Simon W. Moore. *Multithreaded Processor Design*. Kluwer Academic Publishers, June 1996.

46. Richard Murphy. Design parameters for distributed PIM memory systems. Master's thesis, University of Notre Dame, South Bend, IN, April 2000.

47. Richard Murphy. *Travelling Threads: A New Multithreaded Execution Model*. PhD thesis, University of Notre Dame, Notre Dame, IN, June 2006.

48. Richard C. Murphy, Peter M. Kogge, and Arun Rodrigues. The characterization of data intensive memory workloads on distributed PIM systems. *Lecture Notes in Computer Science*, 2107:85–??, 2001. URL `http://citeseer.ist.psu.edu/murphy00characterization.html`.

49. C. Natarajan, S. Sharma, and R. K. Iyer. Measurement-based characterization of global memory and network contention, operating system and parallelization overheads. In *ISCA '94: Proceedings of the 21ST annual international symposium on Computer architecture*, pages 71–80, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5510-0. doi: http://doi.acm.org/10.1145/191995.192018.

50. NeoMagic. Neomagic products, 2001. URL `http://www.neomagic.com`.

51. Yasuhiro Nunomura, Toru Shimizu, and Osamu Tomisawa. M32R/D-integrating dram and microprocessor. *IEEE Micro*, 17(6):40–48, 1997. ISSN 0272-1732. doi: http://dx.doi.org/10.1109/40.641595.

52. Ron A. Oldfield, David E. Womble, and Curtis C. Ober. Efficient parallel I/O in seismic imaging. *The International Journal of High Performance Computing Applications*, 12(3):333–344, Fall 1998. URL `ftp://ftp.cs.dartmouth.edu/pub/raoldfi/salvo/salvoIO.ps.gz`.

53. Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: a computation model for intelligent memory. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, pages 192–203, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8491-7. doi: http://doi.acm.org/10.1145/279358.279387.

54. Jerry Owens. The ASCI sPPM benchmark code readme file, 1996. URL `http://www.llnl.gov/asci_benchmarks/asci/limited/ppm/sppm_readme.html`.

55. David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, 1997. ISSN 0272-1732. doi: http://dx.doi.org/10.1109/40.592312.

56. David A. Patterson and John L. Hennessy. *Computer organization & design: the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1-55860-281-X.

57. Jack Perdue. Predicting performance on SMPs. a case study: The SGI power challenge. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 729, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0574-0.

58. Steve J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117:1–19, 1995.

59. Steve J. Plimpton. LAMMPS web page, August 2006. URL `http://www.cs-sandia.gov/~sjplimp/lammps.html`.

60. Steve J. Plimpton, R. Pollock, and M. Stevens. Particle-mesh Ewald and rRESPA for parallel molecular dynamics simulations. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.

61. Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 3–13, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press. ISBN 1-58113-016-3.

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

63. Arun Rodrigues, Richard Murphy, Peter Kogge, and Keith Underwood. Characterizing a new class of threads in scientific applications for high end supercomputers. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 164–174, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-839-3. doi: http://doi.acm.org/10.1145/1006209.1006234.

64. Saul Rosen. Electronic computers: A historical survey. *ACM Computing Surveys*, 1(1):7–36, 1969. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/356540.356543.

65. Subhash Saini and Horst D. Simon. Applications performance under OSF/1 AD and SUNMOS on Intel Paragon XP/S-15. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 580–589, New York, NY, USA, 1994. ACM Press. ISBN 0-8186-6605-6. doi: http://doi.acm.org/10.1145/602770.602868.

66. A. G. Salinger, R. P. Pawlowski, J. N. Shadid, B. van Bloemen Waanders, R. Bartlett, G.C. Itle, and L. Beigler. Reacting flows optimization with rSQP and MPSalsa. In Ghattas, van Bloemen Waanders, Biegler, and Heinkenschloss, editors, *Proceedings of the First Sandia Workshop on Large-Scale PDE Constrained Optimization*. Springer, 2001. In Lecture Notes in Computer Science.

67. A. Saulsbury, F. Pong, and Andreas Nowatzyk. Missing the memory wall: The case for processor/memory integration. In *International Symposium on Computer Architecture*, May 1996.

68. 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 *Proceedings of the Intel Supercomputer User Group Conference*, 1995.

69. Jaswinder Pal Singh, Edward Rothberg, and Anoop Gupta. Modeling communication in parallel algorithms: a fruitful interaction between theory and systems? In *SPAA '94: Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, pages 189–199, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-671-9. doi: http://doi.acm.org/10.1145/181014.181329.

70. Anand Sivasubramaniam, Aman Singla, Umakishore Ramachandran, and H. Venkateswaran. An approach to scalability study of shared memory parallel systems. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 171–180, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-659-X. doi: http://doi.acm.org/10.1145/183018.183038.

71. Byoungro So, Mary W. Hall, and Heidi E. Ziegler. Custom data layout for memory parallelism. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 291, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

72. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-411-8. doi: http://doi.acm.org/10.1145/383059.383071.

73. J. Tao, W. Karl, and M. Schulz. Memory access behavior analysis of NUMA-based shared memory programs, 2001. URL `citeseer.ist.psu.edu/tao01memory.html`.

74. J. Tao, W. Karl, and M. Schulz. Using simulation to understand the data layout of programs, 2001. URL `citeseer.ist.psu.edu/tao01using.html`.

75. O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE confer-*

ence on Supercomputing, pages 578–587, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. ISBN 0-8186-2630-5.

76. O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 261–271, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-659-X. doi: http://doi.acm.org/10.1145/183018.183047.

77. Olivier Temam, Elana D. Granston, and William Jalby. To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Supercomputing*, pages 410–419, 1993. URL `http://citeseer.ist.psu.edu/temam93to.html`.

78. M. R. Thistle and B. J. Smith. A processor architecture for Horizon. In *Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 35–41, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press. ISBN 0-8186-0882-X.

79. Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.612254.

80. Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/216585.216588.

81. Hui Zhang, Ashish Goel, and Ramesh Govindan. Incrementally improving lookup latency in distributed hash table systems. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 114–125, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-664-1. doi: http://doi.acm.org/10.1145/781027.781042.

This document was prepared & typeset with LaTeX 2ε*, and formatted with* NDdiss2ε *classfile (v3.0[2005/07/27]) provided by Sameer Vijay.*