massachusetts institute of technology — artificial intelligence laboratory

ADAM: A Decentralized Parallel Computer Architecture Featuring Fast Thread and Data Migration and a Uniform Hardware Abstraction

Andrew "bunnie" Huang

AI Technical Report 2002-006 June 2002

ADAM: A Decentralized Parallel Computer Architecture Featuring Fast Thread and Data Migration and a Uniform Hardware Abstraction

by

Andrew “bunnie” Huang

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

Certified by: Thomas F. Knight, Jr.
Senior Research Scientist
Thesis Supervisor

Accepted by: Arthur C. Smith
Chairman, Department Committee on Graduate Students

ADAM: A Decentralized Parallel Computer Architecture Featuring Fast Thread and Data Migration and a Uniform Hardware Abstraction

by

Andrew “bunnie” Huang

Submitted to the Department of Electrical Engineering and Computer Science on May 24, 2002, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

The furious pace of Moore’s Law is driving computer architecture into a realm where the the speed of light is the dominant factor in system latencies. The number of clock cycles to span a chip are increasing, while the number of bits that can be accessed within a clock cycle is decreasing. Hence, it is becoming more difficult to hide latency. One alternative solution is to reduce latency by migrating threads and data, but the overhead of existing implementations has previously made migration an unserviceable solution so far.

I present an architecture, implementation, and mechanisms that reduces the overhead of migration to the point where migration is a viable supplement to other latency hiding mechanisms, such as multithreading. The architecture is abstract, and presents programmers with a simple, uniform fine-grained multithreaded parallel programming model with implicit memory management. In other words, the spatial nature and implementation details (such as the number of processors) of a parallel machine are entirely hidden from the programmer. Compiler writers are encouraged to devise programming languages for the machine that guide a programmer to express their ideas in terms of objects, since objects exhibit an inherent physical locality of data and code. The machine implementation can then leverage this locality to automatically distribute data and threads across the physical machine by using a set of high performance migration mechanisms.

An implementation of this architecture could migrate a null thread in 66 cycles – over a factor of 1000 improvement over previous work. Performance also scales well; the time required to move a typical thread is only 4 to 5 times that of a null thread. Data migration performance is similar, and scales linearly with data block size. Since the performance of the migration

mechanism is on par with that of an L2 cache, the implementation simulated in my work has no data caches and relies instead on multithreading and the migration mechanism to hide and reduce access latencies.

Thesis Supervisor: Thomas F. Knight, Jr.
Title: Senior Research Scientist

Acknowledgments

I would like to thank my parents for all their love and support over the years, and for their unwaxing encouragement and faith in my ability to finish the degree program.

I would also like to thank my wonderful, loving, caring girlfriend Nikki Justis for all her support, motivation, patience, editing, soldering, discussion and idea refinement, cooking, cleaning, laundry doing, driving me to campus, wrist massages, knowing when I need to tool and when I need to take a break, tolerance of my 7 AM sleep schedule, and for letting me make a mess in her room and take over her couch with my whole computer setup.

I would like to thank all my friends for their support over the years, and for making the past decade at MIT–and my first step into the real world–an exciting, fun and rewarding experience. Let the rush begin...and may it never end.

This thesis would never happened if it were not for the Aries Research Group (in order of seniority): Tom Knight, Norm Margolus, Jeremy Brown, J.P. Grossman, Josie Ammer, Mike Phillips, Peggy Chen, Bobby Woods-Corwin, Ben Vandiver, Tom Cleary, Dominic Rizzo, and Brian Ginsburg. Tom Knight, in particular, has been a role model for me since I came to the lab; he is an endless source of inspiration and knowledge, and has provided invaluable guidance, counsel and encouragement. He is brilliant and visionary, yet humble and very accessible, and always willing to answer my questions, no matter how silly or stupid. I also really enjoy his *laissez-faire* policies with respect to running the group; I truly treasure the intellectual freedom Tom brought to the group, and his immense faith in all of our abilities to manage and organize ourselves, and to “go forth and think great thoughts.” Jeremy Brown and J.P. Grossman were also invaluable for their good ideas, lively

conversation, and idea refinement. Jeremy invented the idempotent network protocol used in this thesis, and his excellent thesis work in novel parallel programming methods and scalable parallel garbage collection fills in many crucial gaps in my thesis. J.P. and Jeremy also developed the capability representation with SQUIDS that is central to my thesis. I also relied on J.P.'s excellent work in researching and characterizing various network topologies and schemes; I used many of his results in my implementation. Bobby Woods-Corwin, Peggy Chen, Brian Ginsburg and Dominic Rizzo were invaluable in working out the implementation of the network. Without them, I would have nothing to show for this thesis except for a pile of Java code. Two generations of M.Eng theses and two UROPs is a lot of work! Norm Margolus also helped lay down the foundations of the architecture with his work in spatial cellular automata machines and embedded DRAM processors. Finally, André DeHon, although not officially a part of the group, was very much instrumental to my work in many ways. This work relies very heavily upon his earlier work at MIT on the METRO network. André also provided invaluable advice and feedback during his visits from Caltech.

I would also like to give a special thanks to Ben Vandiver. Since the inception of the ADAM and the Q-Machine, Ben has furnished invaluable insight and feedback. The thesis would not be complete without his synergism as a compiler writer, programmer, and high-caliber software hacker. I also thank him for his enthusiasm and faith in the architecture; his positive energy was essential in keeping me from getting discouraged. He not only helped hash out the programming model for the machine, he also wrote two compilers for the machine along the way. He also was instrumental in coding and debugging the benchmarks used in the results section of my thesis.

Krste Asanović and Larry Rudolph were also very important influences on this thesis. Krste is a wellspring of knowledge and uncannily sharp insight

into the even the finest architectural details. Larry opened my mind to the field of competitive analysis and on-line algorithms, something I would never have considered otherwise. I also appreciate the critical review provided by both Krste and Larry.

I would also like to thank my friends at the Mobilian Corporation–in particular, Rob Gilmore, MaryJo Nettles, Todd Sutton, and Rob Wentworth–for their understanding, patience, and support for me finishing my degree.

I thank the Xilinx Corporation for generously donating the many high-end FPGAs and design tools to the project, which were used to implement prototype network and processor nodes. I would also like to thank the Intel Corporation and Silicon Spice for providing fellowships and equipment that enabled me to finish my work. Sun Corporation and Borland also provided me Java and JBuilder for free, but the value of these tools cannot be underestimated. This work was also funded by the Air Force Research Laboratory, agreement number F30602-98-1-0172, “Active Database Technology”.

I could go on, but unfortunately there is not enough space to name everyone who has helped with my thesis. This is to everyone who provided invaluable input and guidance into my thesis–thank you all. I am indebted to the world, and I can only hope to someday make a contribution that is worthy.

Finally, all mistakes in this thesis are mine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> You can't fake memory bandwidth that isn't there.
>
> —*Seymour Cray on why the Cray-1 had no caches*

Most data and thread migration mechanisms to date are slow when compared to other latency management techniques. This thesis introduces an architecture, ADAM, that enables a simple hardware implementation of data and thread migration. This implementation reduces the overhead of migration to the point where it is comparable to other hardware-assisted latency management techniques, such as caching.

Data migration is useful to reduce access latencies in situations where the working set is larger than cache. It is also useful in reducing or redistributing network traffic in situations where hotspots are caused by contention for multiple data objects. Data migration can also be used to emulate the function of caches in systems that feature no data caches.

Thread migration is useful to reduce access latencies in situations where multiple threads are contending for a single piece of data. Like data migra-

tion, it is also useful in situations where hotspots can be alleviated by redistributing the sources and destinations of network traffic. Thread migration is also useful for load-balancing, particularly in situations where memory contention is low.

Data and thread migration can be used together to help manage access latencies in situations where many threads are sharing information in an unpredictable fashion among many pieces of data, as might be the case in an enterprise database application. Data and thread migration can also be used to enhance system reliability as well, if faults can be predicted far enough in advance so that the failing node can be flushed of its contents.

1.1 Contributions

The primary contribution of my thesis is a **fast, low-overhead data and thread migration mechanism.** In terms of processor cycles, the mechanism outlined in my thesis represents greater than a 1000-fold increase in performance over previous software-based migration mechanisms. As a result, data and thread migration overheads are similar to L2 cache fills on a conventional uni-processor system.

The key architectural features that enable my data and thread migration mechanisms are a **unified thread and data representation using capabilities** and **interthread communication and memory access through architecturally explicit queues**. Threads and data in my architecture, ADAM, are accessed using a capability representation with tags that encode base and bounds information. In other words, every pointer has associated with it the region of data it can access, and this information trivializes figuring out what to move during migration. Architecturally explicit queues, on the other hand, simplify many of the ancillary tasks associated with migrating threads and

data, such as the movement of stacks, the migration and placement of communication structures, concurrent access to migrating structures, and pointer updates after migration.

My thesis also describes an **implementation outline of ADAM** dubbed the "Q-Machine". The implementation technology is presumed to be 35 nm CMOS silicon, available in volume around 2010, and features no data caches; instead, it relies on the migration mechanism and multithreading to maintain good performance and high processor utilization. The proposed implementation is simulated with the ADAM System Simulator (ASS); it is this simulator that provides the results upon which the ADAM architecture is evaluated. Note that there is no requirement for advanced technology to implement the ADAM; one could make an ADAM implementation today, if so desired. The 2010 technology point was chosen to evaluate the ADAM architecture because it would match a likely tape-out time frame of the architecture's implementation.

1.2 Organization of This Work

Chapter 2, "Background", discusses some of the advantages and disadvantages of a migration scheme over more conventional latency management schemes. It also reviews, at a high level, some of the problems encountered in previous migration schemes; a more detailed review of migration mechanisms is presented in Chapter 4. Chapter 2 closes with a differentiation of this work from its predecessors in a brief discussion of the architectural pedigree of the ADAM and its Q-Machine implementation.

Chapter 3, "Aries Decentralized Abstract Machine", describes the ADAM in detail. This chapter lays the foundation for the programming model of the ADAM through a simple code example, followed up with a discussion of

the architectural details relevant to a migration implementation. A detailed discussion of other architectural features can be found in Appendix B.

Chapter 4, "Migration Mechanism in a Decentralized Computing Environment", presents the implementation of the migration mechanisms. The chapter begins with a survey of previous work involving data and thread migration; this survey includes both mechanisms and migration control algorithms, since their implementation details are intimately associated. I then describe the migration mechanism in detail.

Chapter 5, "Implementation of the ADAM: Hardware and Simulation", describes an implementation of ADAM. This implementation is known as the Q-Machine. This chapter summarizes the machine organization and implementation technology assumptions of the simulator used to evaluate my migration mechanisms.

In the next chapter, "Machine and Migration Characterization" (Chapter 6), I characterize the performance of the implementation. The chapter starts with two simple micro-kernel benchmarks and some formal analysis of the migration mechanism. Then, I present results for some more comprehensive benchmarks, Quicksort, Matrix Multiply and N-Body, with simple migration control heuristics driving the migration mechanisms.

The thesis concludes in chapter 7 with a discussion of further developments for the ADAM architecture, areas for improvement and further research, and programming languages for the machine. Note that while a detailed discussion of programming languages for the ADAM is outside the scope of this thesis, I did not work in a programming language vacuum. A strong point of using an abstract machine model is that compiler writers can begin their work on day one, and in fact, that is the case. Benjamin Vandiver, an M.Eng student in my research group, has developed two languages, *Couatl* and *People*, and compilers for these languages to the ADAM architecture.

Couatl is a basic object-oriented language that we used in the early stages of architecture development to hammer out the abstract machine model and to determine the unique strengths and weaknesses of a queue based architecture. The follow-on language, People, is a more sophisticated language supporting streaming constructs that leverages the availability of architectural queues at the language level. I refer interested readers to his M.Eng thesis [Van02].

A summary of the abstraction layers employed by this thesis can be found in figure 1.1. ADAM is a pure abstraction, a boundary between compilers and hardware. Q-Machine is the implementation of ADAM that realizes the fast data and thread migration mechanisms made possible by ADAM. The ADAM System Simulator (ASS) is my software simulation of the Q-Machine, written in Java. The Q-Machine could also be implemented directly in hardware, but that is not within the scope of this thesis.

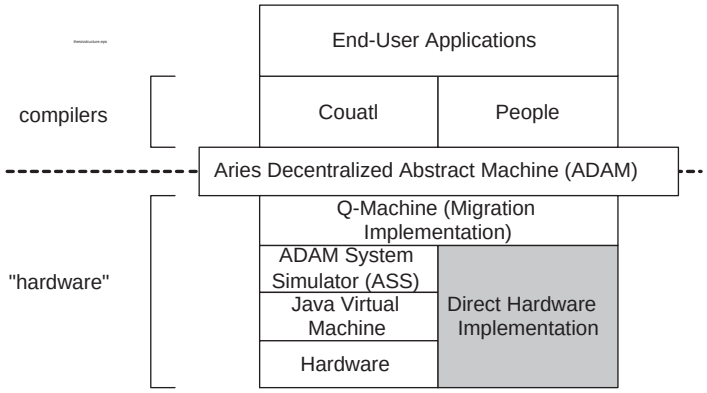


Figure 1.1: Overview of the abstraction layers in this thesis. Couatl and People are compilers written by Ben Vandiver.

I also provide a set of appendices that describe various technical nits of the architecture, including the bit-level details of the ADAM architecture, physical queue file (PQF) implementation, the network interface implementation, network protocols, and opcodes of the ADAM.

Chapter 2

Background

> TSMC sees no insurmountable challenges in the path to scaling [silicon CMOS technology] to the 9 nm node. The question is, will the market be ready for it?
>
> *—Calvin Chenming Hu, CTO of TSMC at a talk at MIT*

This chapter starts by characterizing the ADAM architecture in terms of its use of latency management techniques. This chapter then discusses in greater detail a comparison of various migration techniques. Finally, this chapter closes with a discussion of ADAM's architectural pedigree.

2.1 Latency Management Techniques

Numerous latency management techniques are available to computer architects looking to design large parallel machines. Latency management techniques can be divided into two broad categories, latency reduction, and latency hiding.

2.1.1 Latency Reduction

Latency reduction techniques include architectural trade-offs to optimize local memory access latency, such as non-uniform memory access (NUMA) and cache-only memory architecture (COMA). NUMA architectures cope with the spatial reality of large machines explicitly; thus, local memory references are faster than remote memory references. This is in contrast with bus-based architectures that have uniform memory access times. NUMAs typically employ spatial interconnection networks that are inherently more scalable than bus-based architectures. While NUMAs enable better scalability, they are confronted with the issue of how to arrange data so that optimal performance is achieved. One popular method of addressing the data placement issue is to use a directory-based cache coherence mechanism. Examples of cache-coherent NUMAs (ccNUMAs) include Stanford's DASH [LLG+92], and MIT's Alewife [ABC+95]. COMAs, on the other hand, feature automatic data migration through the use of "attraction memories". COMAs also employ spatial interconnection networks that feature non-uniform memory access times, but in a COMA, memory has no home location. Data migrates in a cache-coherent fashion throughout the machine to their points of access. COMAs have the disadvantage of extra hardware complexity, but have an advantage over NUMA machines when the working set of data is larger than the NUMA's cache size. The ADAM architecture is similar to a COMA architecture, except that ADAM also features thread migration, and that there are no caches–in other words, there can be only one valid copy of a piece data in the machine. Removing cache semantics from memory reduces the hardware requirements, but causes ADAM to lose the benefit of automatic data replication. ADAM attempts to compensate for this loss by providing a hardware-recognized immutable data type that is write-once and can be freely copied throughout the machine. Thread migration also helps compensate for

this loss by allowing threads that contend heavily for a single piece of memory to migrate toward the contested memory location.

Latency reduction can also be applied at a lower level, through migration, replication, scheduling, placement and caching. Replication is a property inherent in cache-coherent memory systems where memory can be marked as exclusive or read-only, and several copies can exist throughout the machine to reduce the perceived access latency at multiple nodes. As mentioned previously, ADAM provides only limited support for data replication. Scheduling and placement are predictive techniques that attempt to reduce latency and balance loads by allocating memory and scheduling threads to be near each other. Scheduling and placement can be either directed explicitly by a programmer, inferred and statically linked in by a compiler, or directed by an intelligent runtime system. Scheduling and placement are important latency reduction techniques in any architecture, but are outside the scope of my thesis. A thorough discussion and comparison of migration techniques is reserved for later in this chapter and in chapter 4.

Caching is perhaps the most widely used latency reduction mechanism. Caches reduce memory latency by keeping the most recently accessed values in a fast memory close to the processor. Caches rely on the statistically good spatial and temporal locality characteristics of data accesses found in most programs. Caches also rely on exclusive ownership of data; since a copy is made of data in main memory, a coherence mechanism is required for correct program execution in an environment where concurrent modification is a possibility. This coherence mechanism can present a challenge when scaling up to very large multiprocessor machines. In particular, simple directory-based or snoopy coherence mechanisms show poor scalability. Snoopy coherence mechanisms are used in bus-based multiprocessors, and suffer from bandwidth limitations due to excess coherence traffic as systems

scale in size. Directory-based protocols are more scalable, but they also have their limits. With a 64-byte block size, a simple directory-based cache coherence protocol has a memory overhead of over 200% for a 1024-processor system [CS99], p.565. Techniques such as limited-pointer schemes [ASHH88], extended pointer schemes [ALKK91], and sparse directories [GWM90] can all be used to mitigate the overhead of cache coherence in large parallel systems, but at the cost of more complex protocols or the need for special mechanisms to handle corner cases where the protocol breaks down. The other problem with caches is that technology scaling is not ideal; buffered wire delays have been rising slightly faster than expected, and the expected capacity of caches per access time is anticipated to decrease as process technologies progress [AHKB00] [McF97]. Figure 2.1 illustrates the fallout of non-ideal wire delay scaling. Since the ADAM architecture already features data migration for latency reduction and can tolerate more access latency due to its use of multithreading and decoupling, no data caches are used in the ADAM implementation outlined in this thesis. The elimination of data caches alleviate the scaling concerns of data caches, and it also helps relieve some of the access time pressure resulting from technology constraints. The down-sides of this decision include slower single-threaded code execution and the loss of automatic data replication inherent in cache coherence schemes. Note that the ADAM implementation, as previously mentioned, compensates for this loss of data replication in part by providing an immutable data type, and in part by migrating threads toward heavily contested memory locations.

2.1.2 Latency Hiding

Latency hiding techniques include prefetching, decoupling, multithreading, relaxing memory consistency, and producer-initiated communication.

Prefetching is the use of predictive mechanisms, either automatic or ex-

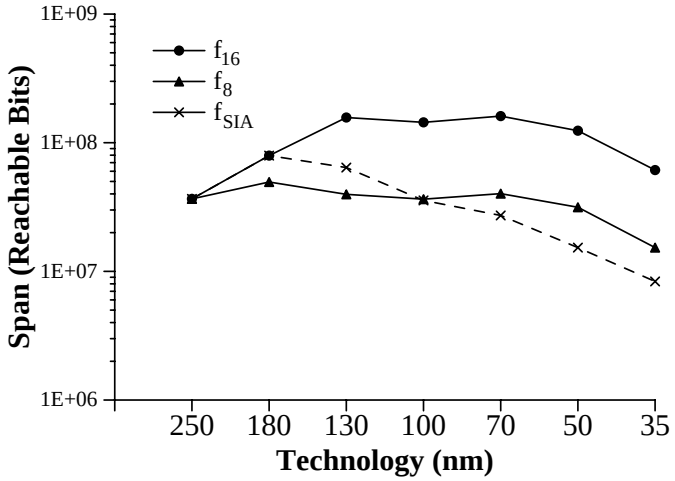


Figure 2.1: Reachable chip area in top level metal, where area is measured in six-transistor SRAM cells. Directly from [AHKB00]

plicit, to access data before a computation requires the data. The efficacy of prefetching is proportional to the accuracy of the predictive mechanism. When the predictive mechanism is wrong, the system can potentially pay a high cost, because improperly prefetched data could displace useful data while consuming bandwidth that could be used for other useful work. Prefetching can be applied in the ADAM architecture, but its implementation is beyond the scope of this thesis.

Decoupling is the use of explicit queues to hide access or compute latencies. Decoupling is featured in decoupled access-execute (DAE) machines, such as the ZS-1 [SDV+87], the WM architecture [Wul92] and the MT-DCAE [SKA01]. Decoupled architectures can be thought of as a type of programmed prefetch architecture, although the decoupling mechanism can also be used to decouple control flow events as well. In a simple DAE architecture, processors are divided into access and execute units, coupled by a set of queues. The access unit is allowed to "slip" ahead of the execute unit, ef-

fectively prefetching data for the execute unit. Since the ADAM uses explicit queues to communicate with threads and to access memory, ADAM shares many of the benefits and problems of DAE architectures.

Multithreading is the use of multiple thread contexts and a fast context switching mechanism to hide memory access latencies. When one thread context stalls on a dependency that requires a lengthy memory access, another thread context is swapped in, thus maintaining a high level of processor utilization. However, multithreading can only effectively hide memory latency if there are enough runnable contexts. As latencies increase, more parallelism is required. The HEP [Smi82a] and TERA [AKK+95] architectures apply multithreading to hide access latencies; the ADAM architecture uses this technique as well.

Relaxed memory consistency models and producer-initiated communication are architectural and programmer-level methods for hiding latency. Relaxing memory consistency models hides latency by allowing systems greater flexibility in hiding write latencies [LW95]. The choice of memory consistency model has a great impact on how a machine is programmed (or compiled to). The ADAM uses a weak ordering model [DS90] similar to that employed in the Alpha [CS99]. Of course, each thread is guaranteed that writes and reads complete in program order on the ADAM as well. Producer-initiated communication reduces latency by cutting out one half of a round trip when the producer and consumer relationships are well-defined. Instead of a consumer sending a message to request data and waiting for the response, producer-initiated communication pushes data into a consumer's cache or queue. In a cache-coherent system, this can lead to higher coherence traffic because all shared copies have to be updated on every write [LW95]. In ADAM, producer-initiated communication is the only mode of communication when using mapped queues. There is no coherence overhead for this

style of communication in ADAM because the queue namespace is separate from memory namespace, and all queue mappings are exclusive by definition.

2.2 Migration Mechanisms

Migration mechanisms tend to be tailor-made to a particular architecture, operating system, or application. As a result, the features of migration schemes are equally diverse. For example, in a network-of-workstations (NOW), migration mechanisms tend to operate on coarse-grained processes and objects. Migration on NOWs tend to be under dynamic run-time control, and migration times are on the order of tens to hundreds of milliseconds. [RC96] On the other hand, computation migration on Alewife [HWW93] implements structured activation frame movement throughout the machine using statically compiled migration directives, yielding migration times on the order of several hundreds of processor cycles.

At the least common denominator, every migration mechanism must do the following things: figure out what to move, prepare the receiver, send the data, and then handle any forwarded requests or pointer updates. Thread or process migration schemes also have to handle task scheduling issues as well. Process migration in NOWs is incredibly inefficient and slow because the abstraction boundary for processes is too high; for example, moving a process entails creating a virtual address space and moving file handles. [RC96] introduces a faster, more streamlined version of process migration that removes the restriction that communication producers be frozen during consumer migration (*i.e.*, enables concurrent communication during migration), but even then process migration takes 14 ms. [CM97] also introduces faster techniques for dealing with pointer updates after migration using explicitly managed pointer registries. The problem with explicitly managed pointer registries, however,

is that incorrect program execution results if the programmer forgets to register a pointer. DEMOS/MP [PM83], interestingly, is a multi-processor operating system introduced over a decade before either [RC96] or [CM97], and it features automatic pointer updating and concurrent communication during process migration. DEMOS/MP features explicit OS-managed communication queues for inter-process communication; this helps enable concurrent communication during process migration and simplifies pointer updates because the migration manager does not have to make guesses or conservative assumptions about the process communication mechanism. Unfortunately, the DEMOS/MP paper contains little performance information on its process migration mechanism, so it is more difficult to compare DEMOS/MP against other works. The ADAM thread migration mechanism implements many features of the DEMOS/MP migration mechanism, except at a finer grain and with hardware support.

On SMP-type machines, migration times are shorter, thanks to the tighter integration of network interfaces and processors, generally faster interconnection networks, finer granularity of objects, and globally shared system resources. Page migration in DASH, for example, takes 2 ms (about 66,000 memory cycles) [CDV+94]. This does not include the time spent waiting for locks in the kernel's virtual memory system; the paper indicates that the response time for workloads were not improved because of this overhead. Even if one could migrate threads in DASH by simply throwing a program counter over the fence to another processor, the overhead of migrating the thread's associated process state–the stack and heap–would be fairly large, since at least two memory structures have to be moved, perhaps at the page level of granularity. Thus, the thread scheduler should be aware of a task's memory footprint, and use cache affinity scheduling to achieve good performance. [CDV+94]

Active Threads [WGQH98] introduces user-space thread migration, so as to bypass the overhead of migrating kernel threads. In addition, Active Threads uses simple user-space messaging protocols for communication, to cut the overhead of copying messages and buffers in OS space. User-space thread migration reduces thread migration latencies down to about 150 μs (about 16,000 processor cycles). Computation Migration [HWW93] also performs users-space thread migration, but in a more restrictive fashion. In Computation Migration, static annotations in user code cause a thread to spawn new procedures on remote nodes; also, [HWW93] makes no indication that inter-thread communication resources are migrated. A single thread thus snakes its way through the machine, with a trajectory that tracks the location of the working set of data. Computation Migration is fast, as it requires only 651 cycles to start a new thread on a remote processor. Even so, a breakdown of the costs of Computation Migration indicate that a large amount of time is spent in procedure linkage, thread creation, and marshaling thread state. As a side note, Computation Migration is not used as the comparative benchmark for ADAM's migration mechanism because Computation Migration implements a restricted version of thread migration that does not accommodate the level of dynamism or concurrency found in the next fastest migration implementation, Active Threads. Hence, Active Threads is used as the comparison point for ADAM's migration mechanism.

Note that this brief review of migration mechanisms is expanded upon in the background section of chapter 4.

2.2.1 Discussion

The ADAM architecture structures threads, data, and their communication mechanisms in such a way as to eliminate or drastically reduce the overheads experienced by the migration mechanisms outlined above. For exam-

ple, almost all migration mechanisms have to deal with pointer updates and message forwarding. The issue is that interthread communications almost always use memory resources, so that any thread migration requires movement of stacks, OS structures, or heap-allocated communication structures. The ADAM architecture condenses communication structures into explicitly named resources through the use of explicit queues. As a result, communication state is stored as part of thread state, and migration of a thread typically involves a single copy operation. The use of bounded capabilities to represent a thread's state in memory, as well as all heap data structures, also simplifies migration, because the region of memory to be copied during migration can be directly computed given a pointer to a thread or data object. The use of bounded capabilities also offers more flexibility in the choice of migration granularity when compared to schemes that require page-level migration, such as that used in DASH [CDV+94]. Another benefit of bounded capabilities is that false data sharing is not possible. For example, in a conventional system two objects can, by random chance, share a cache line or a page of memory (see figure 2.2). If the two memory objects are concurrently accessed by threads on different nodes, the cache line or page of memory will either end up ping-ponging between the nodes, or one thread will have to suffer unfair access times. On the downside, bounded capabilities does not help when a programmer writes code that that explicitly shares objects among many scattered threads. In this case, thread migration should be used to minimize access latencies.

2.3 Architectural Pedigree

The genesis of the ADAM architecture lies in the Dataflow architectures, Decoupled-Access/Execute (DAE) architectures, Processor-In-Memory (PIM)

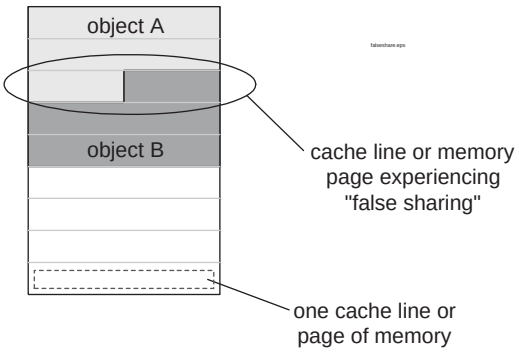


Figure 2.2: Illustration of the false sharing problem.

and Chip Multi-Processor (CMP) architectures, and Cache Only Memory Architectures (COMA).

2.3.1 Dataflow

ADAM is perhaps most closely related to the dataflow family of architectures, in particular, *T. Hence, a careful examination of the dataflow machines is important at this time.

Dataflow machines are a direct realization of dataflow graphs into computational hardware. Arcs on a dataflow graph are decomposed into tokens. Each token is a continuation; it contains a set of instructions and its evaluation context. The length of the instruction run and evaluation context method encapsulated within a token can characterize the spectrum of dataflow architectures. In the MIT Tagged-Token Dataflow Architecture (TTDA), each token represents roughly one instruction and its immediate dependencies and results, and token storage is managed implicitly. TTDA evolved into the Monsoon architecture, which has explicit evaluation context management and single-instruction tokens. With Monsoon, tokens contained a value; pointers to an instruction, and pointers to evaluation contexts that are compiler-

generated frame allocations in a linearly addressed structure. Monsoon evolved into the P-RISC and *T architectures, which are machines with tokens that effectively refer to instruction traces and relatively large ”stack-frame” style explicitly allocated frames. The tokens in P-RISC and *T carried only an instruction pointer and a frame pointer, as opposed to any actual data [AB93] [NA89]. One could take this one step further and claim that a Simultaneous Multithreading (SMT) architecture is a dataflow machine with as many tokens as there are thread contexts, and that a conventional Von Neumann architecture is a single-token dataflow machine. [LH94] provides an excellent overview of dataflow machines and an analysis of their shortcomings.

Dataflow machines, while elegant, have a few fatal flaws. Their evolution from the TTDA into near-RISC architectures provides a clue into what these flaws are. The rather abstract TTDA decomposed dataflow graphs to a near-atomic instruction level. Thousands of tokens are created in the course of even a simple program execution, because tokens can be formed and dispatched before dependencies are resolved. [AB93] states that “these tokens represent data local to inactive functions which are awaiting the return of values undergoing computation in other functions invoked from within their bodies”. The execution of any token required an associative search across the space of all tokens for the tokens that held the results that satisfied the current token’s data dependencies. The multi-thousand element associative structure required to do this search is not implementable even after twenty years of process scaling.

Another flaw of the early Dataflow machines is that every token represents a high-overhead synchronization event. [Ian88] points out that von Neumann architectures also perform a synchronization event between each instruction, but the method of synchronization is very light-weight: IP = IP + 1 or IP = branch target. This allows von Neumann architectures to grind

through straight-line code very quickly. Fortunately for the von Neumann crowd, most code written to date can be straightened out sufficiently with either branch prediction or trace scheduling to get good performance out of such a system. P-RISC and *T leveraged this strength of von Neumann architectures somewhat by allowing a token to represent what are essentially an execution trace and a stack frame. *T actually has a very similar single-node architecture to the ADAM: it divides a single node into a synchronization coprocessor and a data processor. The synchronization processor is responsible for scheduling threads and dealing with synchronization issues, while the data processor's exclusive job is to execute straight-line code efficiently. However, the similarity ends there, as the *T architecture focuses primarily on latency hiding through rapid and efficient thread scheduling, starting, and context switching. While latency hiding through multithreading is an important part of the ADAM architecture, it is also very important to reduce latency by providing mechanisms for the efficient migration of data and threads between processor nodes. The ADAM's overall organization reflects this attention to migration mechanisms. Also, a careful examination of the implementation strategy outlined in [PBB93] reveals a number of important differences (and similarities) between the ADAM and *T. One significant difference is ADAM's use of a queue-based interface between threads, with implicit synchronization through empty/full bits, similar to the scheme used in the M-Machine [FKD+95]. *T uses a register-based interface with a microthread cache to enable efficient context switching, and explicit, program-level handling of messages that could not be injected into the network. The use of self-synchronizing queues of an opaque depth in ADAM helps cushion network congestion and scheduling hiccoughs.

2.3.2 Decoupled-Access/Execute

Decoupled-access/execute (DAE) machines are single-node processors with separate execute and access engines. These engines are coupled with architecturally visible queues that are used to hide memory access latencies. Code for these machines are typically broken down by hand or compiler into an access and execute thread; latencies are hidden because the access thread, which handles memory requests, can ”slip” ahead of the execute thread. Relatively few machines have been built that explicitly feature DAE. The architecture was first proposed in [Smi82b] and later implemented as the Astronautics ZS-1 [SDV+87]. [MSAD90] characterizes the latency-hiding performance of the ZS-1 in detail, and [MSAD91] compares the performance of the ZS-1 to the IBM RS/6000. A comparison of DAE versus superscalar architectures can be found at [FNN93], and a comparison of DAE versus VLIW architectures can be found at [LJ90]. Another proposed DAE architecture is the WM Architecture [Wul92], and a novel twist on DAE architectures where the access unit is actually co-located with the memory is proposed in [VG98]. The architecture described in this work parallels many of the ideas in [VG98].

The basic message contained in all the previously cited papers is that by judiciously dividing a processor into two spatially distributed processors, greater than 2x performance gains can be realized. This super-linear speedup results from latency that was architecturally bypassed by either allowing the memory subsystem to effectively slip ahead and prefetch data to the execution unit, or by physically co-locating the access unit with the memory. DAE ideas can actually be applied generically to any machine with a large amount of explicit parallelism by simply dividing every program into two threads, an access thread and an execute thread. The advantage of explicit DAE machines is that the synchronization between the access and execute threads is very fast because they are coupled via hardware queues, as opposed to software emu-

lated queues. Some conventional out of order execution machines also provide a certain amount of implicit access/execute decoupling via deep, speculative store and load buffers. However, in general, conventional architectures that emulate these queues in software pay a high price for synchronization overhead. Software implementations that use polling to check empty bits pay the overhead of polling plus any time lost between the actual data availability event and the poll event. Interrupt-driven implementations are also expensive because typical interrupt mechanisms require kernel intervention.

Another important message is that queues are like bypass capacitors for computer architectures. Queues low-pass filter the uneven access patterns of high-performance code and help decouple the demand side of a computation from the supply side of a computation. Like bypass capacitors, the time constant of the queue (*i.e.*, the size of the queue) has to be sufficiently large to filter out the average spike, but not so large as to reduce the available signal bandwidth and hamper important tasks such as context switching. The overhead of the queue structure must also be small so that the benefits of queuing can be realized.

Unfortunately, simple DAE machines as a whole suffer from a few problems. There are no compilers that generate explicit access and execute code streams; most benchmarks and simulations in the cited papers were with hand-coded access and execute loops. Also, the effectiveness of DAE is questionable on complicated loops and programs with complicated and/or dynamic dataflow graphs. Simple DAE is targeted at hiding memory latencies, and not much else. However, the basic idea of decoupling access and execute units is a powerful one; especially if the physical access and execute units are allowed to be assigned dynamically to a single virtual control thread, as is the case in ADAM. Creating these "virtual" DAE machines allows access and execute units to migrate throughout the machine and optimize latency on a

thread by thread basis. A sufficiently flexible infrastructure would also allow several execute units to be chained together, thus providing a kind of loop unrolling and a facility for streaming computations without any modification to the code. Because this chaining is dynamic, such a machine could be upgraded to have more processors and a greater performance would be realized without recompiling the code. This idea of a virtual DAE architecture is an important part of the ADAM architecture.

2.3.3 Processor-In-Memory (PIM) and Chip Multi-Processors (CMP)

Recent advances in process technology have made it possible to integrate a sufficient amount of SRAM on-chip to make a single-chip stand-alone processor node. Also, the availability of DRAM embedded on the same die as a processor opens the door to even higher levels of memory integration [Corb] [Mac00] [Cora]. This integration of processors and memory on a single die is referred to as Processor-In-Memory (PIM). The fact that the memory is included on the same die as the processor implies a power and performance advantage due to the elimination of chip-chip wiring capacitances and wire run lengths. It also offers a performance advantage because more wires can be run between the memory bank and the processor than in a discrete processor-memory solution. As process technology continues to improve, it will be possible to put several processor cores plus memory on a single silicon die. This style of implementation is known as a Chip Multi-Processor (CMP). A paper that summarizes some of the key arguments for CMP architectures can be found in [ONH+96]. Some architectures that have been proposed which take advantage of some combination of embedded memory technology and chip multiprocessor technology include RAW [LBF+98], I-RAM [KPP+97], Active Pages [OCS98], Decoupled Access DRAM [VG98], Terasys [GHI94],

SPACERAM [Mar00], and Hamal [Gro01].

The level of performance available to users of embedded DRAM is remarkable. Traditionally, DRAM is thought of as the sluggish tanker of memory, while SRAM is the speed king. A recent DRAM core introduced by MoSys (the so-called 1-T SRAM), available on the TSMC process, has proven that DRAM has a place in high performance architectures [Cora]. The 1-T SRAM is based on a DRAM technology, but has a refreshless interface like a SSRAM (synchronous SRAM). The performance of this macro is also sufficiently high – 2-3 cycle access times at 450 MHz in a 0.13 μm process – to entirely eliminate the need for data caches in the processor design. Note that the processor frequency targets for ADAM is on par with compiled "soft core" processor frequency targets, which is typically a factor of 2-4 below the level of the full-custom processors developed by Intel, AMD, and Compaq. The ADAM is assumed to be implemented using a portable RTL design flow, optimized for fast design cycles and portability to the latest process technology offered by foundries. The reduced implementation time and the CMP architecture of the ADAM helps compensate for the performance penalty of using a compiled design flow. Finally, because the 1-T SRAM has the memory cell structure of DRAM, the density of these macros is similar to the embedded DRAM macros offered in other processes (2.09 mm^2 per Mbit for a DRAM macro on IBM's Cu-11 process [Mac00] versus 1.9 mm^2 per Mbit for a MoSys macro on a TSMC 0.13 μm logic process [Cora]).

The ADAM architecture leverages both the high level of logic integration available in future process technology and the availability of off-the-shelf, fast, dense memories to create a distributed massively parallel architecture with good single-threaded code performance.

2.3.4 Cache Only Memory Architectures

While the architecture proposed in this thesis has no data caches, one could argue that the speed of the memories used in the processor nodes qualifies them as program-managed caches. Hence, it is important to look at the class of machines known as Cache Only Memory Architectures (COMA). The most relevant machine in this class in the Data Diffusion Machine (DDM) [MSW93]. The DDM relies on data migration through the implicit semantics of caches. Because this work is so closely tied to data migration and its control, a thorough discussion of how ADAM relates to the DDM is deferred until section 4.

Chapter 3

Aries Decentralized Abstract Machine

> While Newton is to have said (sarcastically, in truth, but that's another story) that he saw farther by standing on the shoulders of giants, most of us squat on the kneecaps of pygmies. But that is meant in the nicest possible way.
>
> *—Thomas H. Lee, ISSCC 2002 Panelist Statement*

The Aries Decentralized Abstract Machine (ADAM) is an abstract parallel computer architecture optimized for, among other things, fast data and thread migration. This chapter presents an overview of the architecture, highlighting the salient features that enable the implementation of high performance migration. A simple code example is presented first, to acquaint readers with basic ADAM communication and memory abstractions. The example is followed by a more formal, in-depth discussion of various features of the ADAM.

3.1 Introduction to ADAM by Code Example

ADAM has a fine-grained multithreaded programming model. Inter-thread communication and memory access is accomplished via explicit queue resources. Also, memory is abstract; pointers are represented as capabilities with base and bound tags. Programmers cannot create capabilities; they must request one from the machine via an `ALLOCATE` opcode.

3.1.1 Basics

A simple program example that illustrates the salient features of the architecture can be seen in figure 3.2. This code illustrates procedure linkage, capability allocation, and memory mappings. The basic format of assembly opcodes is `OP qa,qb,qc`, where `OP` is the operation, `qa` and `qb` are the arguments, and `qc` is the result. Every operation may have zero, one or two arguments, and one of the arguments may be a constant. There are also some important opcodes that do not follow this format, such as `MAPQC`, that will be discussed soon. Also note that every queue specifier can be modified with an `@` (copy/clobber) modifier. Figure 3.1 demonstrates the operation of the `@` modifier. On reads, an `@` specifies that the instruction should copy the value from an argument queue, instead of dequeuing it. On writes, an `@` specifies that the instruction should overwrite ("clobber") the newest value in a queue, if there is one, instead of enqueuing a value. If the destination queue is empty, the `@` operator has no effect. The `@` operator is handy when dealing with temporaries that are reused frequently; without it, any time a result is used more than once, the programmer or compiler would have to include a special instruction to duplicate values.

```
MOVEC   2, q0   ; initialize q0 with the number 2
MOVEC   1, q1
MOVEC   4, q1   ; initialize q1 with the numbers 1 and 4
ADD     @q0, q1, q2
  ; at this point, q2 has 3, q0 has 2, q1 has 4
ADD     q0, q1, @q2
  ; at this point, q0 is empty, q1 is empty, and q2 has 6
```

Figure 3.1: Demonstration of the copy/clobber (@) modifier.

3.1.2 Calling Convention

In ADAM, the calling convention is that every procedure is a new thread. Arguments and return values are passed via queue mappings. The code in figure 3.2 demonstrates this calling convention. The caller, `main`, calls `testStub` by executing a `SPAWNC q2,testStub,q0` instruction. This instruction starts a new thread with its program counter set to the label `testStub` and returns the new thread's context ID in `q0`. The argument `q2` is the *spawn metric*; this lets the programmer control the placement of new threads. In this case, the spawn metric was initialized to 1, which causes the new thread to be started on some node one network hop away.

After creating the new thread, the caller maps a queue into the callee's queue space to initiate argument passing. *Mapping a queue* causes values written into the mapped queue to appear eventually in the map target. The storage location of data written into a mapped queue is the map target. Also, communication via queue maps is push-only; one cannot read from a mapped queue. Hence, once a queue is mapped, it is write-only; a read from a mapped queue results in undefined behavior. In this example, the new thread expects all of its arguments in `q0`, so the caller maps to the new thread using the instruction `MAPQC q1,q0,@q0`. Note that the `MAPQC` instruction has unusual semantics. The first two arguments are actually immediate constants; in

```
main:
        MOVECC  1, q2           ; set spawn metric to 1
        SPAWNC  q2,testStub,q0  ; spawn remote thread
        MAPQC   q1, q0, @q0     ; map to my child
        PROCID  q1              ; send my procID to child
        MOVE    q20, q22        ; wait for return val from child
        MML     q40, q41        ; declare q40, q41 as load queues
        MOVE    @q22, q40       ; initialize q40 w/capability
        MOVECL  0, q40          ; retrieve data from offset 0
        PRINTQ  q41             ; print (sim specific instruction)
        HALT
testStub:
        MOVE    q0, q100        ; store caller in q100
        MAPQC   q1, q20, @q100  ; my q1 -> q20 of my caller
        MOVECC  0, q2           ; set allocate metric to 0
        ALLOCATEC q2, 8, q10    ; allocate 8-word local capability
        MMS     q30, q31        ; declare q30, q31 as store queues
        MOVE    @q10, q30       ; init q30 w/capability
        MOVECL  0, q30          ; store data 10 at offset 0
        MOVECL  10, q31
        MSYNC                   ; ensure that store has committed
        MOVE    @q10, q1        ; send the capability to my caller
        HALT
```

Figure 3.2: Simple code example demonstrating procedure linkage, thread spawning, memory allocation, and memory access.

other words, they are interpreted as simply queue numbers, and not as sources for operands. The first value, `q1`, specifies the local queue to be mapped. The second value, `q0`, specifies the queue number of the map target. The final argument, `@q0`, specifies the queue from which to read the map target's context ID. I chose the first two values to be constant values because programmers or compilers typically know exactly what the source and destination queue numbers of a mapping should be.

Now that the caller has mapped the argument queue to the callee, the caller first passes its context ID to the callee. Upon receiving the caller's context ID, the callee maps a return queue back to the caller. In this example, the caller and callee agree by convention that `q20` is the return value queue. Figure 3.3 illustrates the state of the caller and callee after setting up the argument and return queues.

3.1.3 Memory Allocation and Access

The next set of instructions in our code example demonstrate memory allocation and access. Memory allocation in ADAM is accomplished with the `ALLOCATE` instruction, and memory access is accomplished through queue mappings.

In this particular example, the instruction `ALLOCATEC q2,8,q10` is used to create a new capability. `q2` is an allocation metric similar to the spawn metric used by the `SPAWNC` opcode. In this case, `q2` is initialized to 0, so this instruction is requesting the allocation of local memory.

The next instruction, `MMS q30,q31`, declares `q30` and `q31` to be store queues. The arguments to `MMS` are immediate constants, similar to the `MAPQC` instruction. Subsequent to the `MMS` instruction, `q30` is a *store address* queue, and `q31` is a *store data* queue. Data can be stored to memory using this pair of queue mappings by enqueuing address and data pairs into their re-

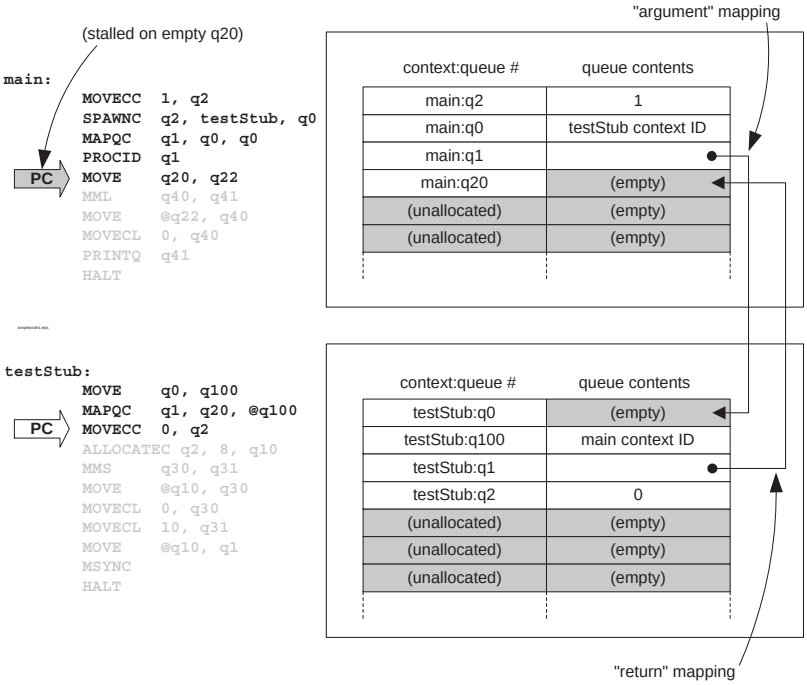


Figure 3.3: Thread states after thread spawn and procedure linkage.

spective queues. Before storing data using these queues, the store address queue must be initialized with a store capability. This is accomplished by the `MOVE @q10,q30` instruction; it copies the allocated capability in `q10` into the store address queue `q30`. Subsequent writes into the store address queue should be constant offsets to the initial capability; the memory subsystem is responsible for adding this offset and checking for bounds violations. Writing another capability into the store address queue causes the store address queue to be re-initialized with the new capability.

In our code example, a single value, 10, is stored at offset 0. The thread `testStub` then performs an `MSYNC` to ensure that the store has committed, and sends the memory capability to the calling thread and halts. The caller, `main`, then establishes load address and load data queues using the `MML q40,q41` instruction. `main` then accesses the returned data capability by sending a copy of the capability into the load address queue, `q40`. `main` then prints the return value from memory and halts. The `PRINTQ` instruction is a convenience instruction only used in the simulator implementation for debugging purposes. The final state of our machine at the end of our code example run is illustrated in figure 3.4.

3.2 Programming Model

This section fleshes out some of the basic architectural features of ADAM presented in the simple code example. For a discussion of architectural features and implementation details not directly relevant to migration, please see appendix B. Things discussed in appendix B include the instruction formats, detailed breakdowns of the capability format bitfields, exception handling, and kernel/OS interactions. For a comprehensive review of the opcodes provided in ADAM, please refer to appendix D.

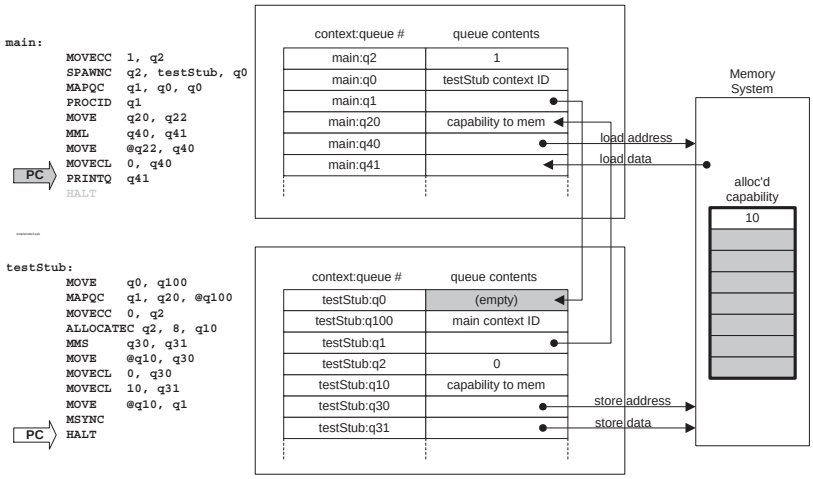


Figure 3.4: Thread states after memory allocation and access.

3.2.1 Threads

The fundamental unit of computation in ADAM is a thread. Threads are very lightweight under ADAM, and they are opaque, monolithic memory structures. They could almost be called continuations except that they carry an activation frame's worth of data in addition to a program counter and an environment pointer. Every thread's state has a one-to-one mapping with a region of memory, as seen before in the named state register file [ND91]. The address and bounds of this region of memory is identified by a capability; this capability is referred to as a thread's *context ID*. Thus, any thread can be globally uniquely identified by its context ID, because the context ID is just a pointer into memory. Also, the number of threads per processor is limited only by the amount of memory available. The correlation of every thread state to a region of memory allows thread and data migration implementations to share the same basic mechanism. A summary of the state associated with a single ADAM thread can be seen in figure 3.5.

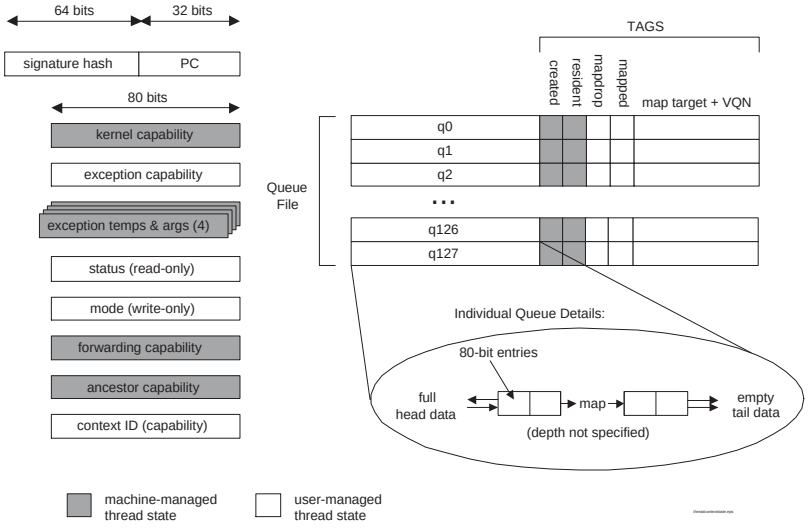


Figure 3.5: Programming model of ADAM

In place of registers in a typical machine, ADAM supplies queues of an unspecified depth. The output of any queue can be remapped onto the input of another queue in another thread context for inter-thread communications. This technique is referred to as *queue mapping*.

Arguments and return values are passed between threads via queue mappings; there is no stack in ADAM. Also, communication to memory is implemented using queue mappings. Hence, all visibility into and out of a thread occurs via a set of queue mappings. This idea is illustrated in figure 3.6. The use of queue mappings simplifies an implementation of thread migration first by isolating all thread state, including communication state, within a single contiguous region of memory, and second by enabling simple mechanisms for managing the forwarding of communications concurrently with migration. These migration mechanisms will be described in chapter 4.

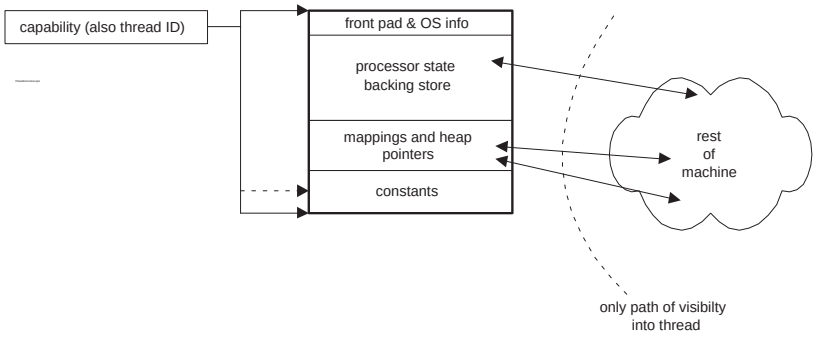


Figure 3.6: Structure of an ADAM thread

3.2.2 Queues and Queue Mappings

To a first approximation, the queues supplied by ADAM are of infinite depth. However, in a realistic implementation, the performance of the queues diminishes as more data is shoveled into them. Hence, while the programming abstraction allows programmers to store large amounts of data in queues, this should be avoided for performance reasons. If a programmer obeys this restriction, the queues should perform comparably to a register in a standard RISC machine (see appendix C for implementation details). Also, when the queues are used as a communication element between streaming threads, flow control is accomplished by applying back-pressure (*i.e.* enqueue stalling) proportional to their fullness. This allows programmers to chain together streaming threads that compute at different rates without having to deal with flow control explicitly.

Queue mapping is the recommended method for inter-thread communication. Data from any given source is guaranteed to arrive in-order in the destination context's queue; however, when more than one sender is mapped to a single receiver, there is no guarantee as to the ordering of the received values between the two senders. A node can request that the source ID of incoming

data be enqueued in a secondary queue in lock-step with the primary destination queue, so that ambiguity created by such a situation can be resolved by user code. While a programmer can communicate data between threads by passing around heap-allocated data structures, it is not recommended because ADAM's memory model uses weak ordering [LW95], and makes no guarantees on the relative ordering of memory requests between threads. Using heap-allocated data structures for inter-thread communication can also be less efficient than direct queue mappings in the presence of thread migration, because heap-allocated communication structures do not automatically migrate with threads.

ADAM queues can assume register semantics when necessary via a copy/clobber modifier, as described in the code example at the beginning of this chapter.

3.2.3 Memory Model

The ADAM uses a virtually addressed capability-based memory model. As mentioned previously, the capability format used in ADAM also encodes base and bound information in the pointer tags. This technique has been seen before in [CKD94], and is refined by [BGKH00]. Capabilities are tagged pointers that the hardware recognizes and treats differently from regular data. In particular, regular users cannot create capabilities on their own; they must request capabilities from the operating system or some other trusted supervisory mechanism. This feature helps make a system more secure against malicious or broken code. In the case of ADAM, the capability format is augmented with tag bits. These tag bits encode information about the capability, such as the read/write permissions and the base/bound information. The base and bound tag information is particularly important toward enabling the implementation of fast migration mechanisms. Given an ADAM capabil-

ity, one can deduce the exact region of data to copy from the base and bound tags; note that the base address given to a user in a capability is allowed to be different from the absolute beginning address of the capability. In addition, the tags include an “increment-only” bit. When this bit is set, users can only reference offsets to the capability base that are positive integers, including zero. This allows the system to hide information at the top of each capability from users, between the absolute capability beginning and the user base address. This feature is used in my migration implementation to associate a remote data locater pointer with each capability. The function of the remote data locater pointer is described in detail in chapter 4. For more information about the implementation of base and bounds encoding in ADAM, readers are referred to appendix B.

Memory is striped across the machine using an explicit node ID as part of the address. The node ID field and address field can steal bits from each other depending upon the implementation parameters. This kind of node location coding within the address has been seen before in the Cray T3E [Sco96]. The actual translation of the virtual addresses and paging mechanisms are transparent to the specification and implementation-specific. A summary of the capability format can be seen in figure 3.7.

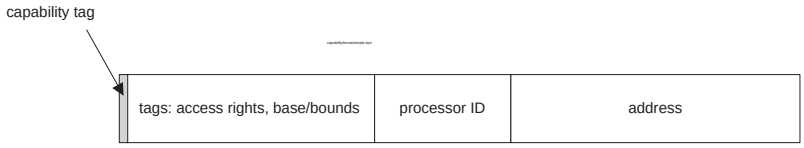


Figure 3.7: High-level breakdown of the ADAM capability format. Detailed bit-level breakdowns of each field can be found in appendix B.

3.2.4 Interacting with Memory

As mentioned previously, there are no load or store instructions in the ADAM specification; memory is an opaque object accessed only through queue mappings. The `MML` and `MMS` opcodes are used to define load and store queue pairs, respectively. `MML` takes an outgoing address queue and a return data queue as arguments; `MMS` takes an outgoing address queue and an outgoing data queue as arguments. The ordering of data in any single given load or store queue mapping within a thread is guaranteed to be preserved, since address and data values are sent to the memory subsystem in lock-step. However, the ordering between multiple sets of mappings is not guaranteed between `MSYNC` instructions. Hence, accessing a single piece of memory through multiple queue maps is not recommended as it can result in nondeterministic behavior.

Locks and semaphores in memory can be implemented using the `EXCH` opcode. The `EXCH` opcode declares a set of three queues as an *exchange tuple*. One queue is used to specify the exchange address, another queue is used as the source of outgoing exchange data, and the final queue is used to specify the return point for the exchanged data. This exchange is guaranteed by hardware in the memory subsystem to be atomic. The timing of the exchange is not deterministic: the actual exchange on the memory location happens whenever the exchange request arrives at the destination memory location.

When initializing a memory queue mapping, the first piece of data written into an *address* queue must be a capability or a memory access exception is thrown. Subsequent accesses to an address queue may pass more capabilities or any integer data type. When an integer data type is put into a memory queue, it is assumed to be an offset of the most recent capability passed into the address queue. Putting a packed integer into an address queue causes data to be returned for each of the packed sub-values, starting with the least

significant value and ending with the most significant value.

A feature of the memory queue access form is that architects and implementers can extend the ADAM specification by adding intelligence to the memory system. Capabilities and offsets are thrown into a memory queue, and the memory system is free to do what it likes before returning some data. Thus, the memory system can be augmented to be more than just a table of stored values; it could be configured to perform computations or to automatically traverse data structures as well.

Chapter 4

Migration Mechanism in a Decentralized Computing Environment

> Memory is like an orgasm. It's a lot better if you don't have to fake it.
>
> —*Seymour Cray on virtual memory*

4.1 Introduction

The idea of moving code and data around so that they are physically closer to each other is appealing in any computer system where communication latencies are high. Unfortunately, migration introduces a large number of new problems. First and foremost, migration consumes computing resources, and system architects must contend with the fact that any movement of data must

be eventually amortized by the resulting reduction in communication latency. The overhead of a migration mechanism includes not only the time to copy the data, but also the time required to negotiate with the migration destination; the potential stalling of access to the data during the migration interval; the time required to update any pointers into the migrated memory; and any collateral impact on network and CPU utilization. This litany of performance pitfalls makes it very difficult to wedge an effective migration mechanism into an existing architecture that was designed without any thought toward the problem. Thus, even though data and thread migration seem to be good ideas in principle, their implementation can be a difficult task.

The ADAM architecture and its corresponding implementation drastically reduce the overhead required for data and thread migration when compared to traditional architectures. ADAM's data and thread migration mechanisms are basically identical because of its programming model and implementation: threads are just data structures that have a special meaning to the thread scheduler. Inter-thread and memory communication is explicitly managed so implementing forwarding pointers and pointer updates can be done through an efficient and straightforward scheme called "temporally bidirectional pointers". Finally, the use of a capability-based memory system with tag-encoded explicit base and bounds on memory regions simplifies the bookkeeping on which pieces of memory to move. It now becomes reasonable to discuss a whole new set of issues related to the on-line scheduling of data and thread migration because of this low-overhead migration mechanism.

4.2 Background

This background section surveys the mechanisms and algorithms of previous work in the area of data and thread migration. This section is divided into

architecture, mechanisms, and algorithms sections.

4.2.1 Architectures that Directly Address Migration

There are a few architectures that directly address data or thread migration. A class of architectures known as COMA (Cache Only Memory Architecture) must grapple head-on with the issue of data migration as a cache line placement problem. NUMA (Non-Uniform Memory Access) machines also introduce the idea of spatial awareness to an architecture, but the issue of data migration is typically encapsulated by the cache coherence protocol. Thread migration mechanisms, on the other hand, typically do not manifest themselves as architectural features, but as run-time or compile-time supported features of otherwise conventional parallel architectures. Therefore, in the literature, thread migration mechanisms typically fall under the genre of work-stealing and load-balancing mechanisms and are treated that way in the next section.

There are relatively few COMAs in the literature. The most notable COMAs are Bristol's Data Diffusion Machine (DDM) [MSW93], the Kendall Square Research KSR-1 [ea92], and the UIUC Illinois Aggressive COMA (I-ACOMA) [TP96]. All three COMAs listed here rely upon a directory-based cache coherence scheme. The KSR-1 and later revisions of the DDM employ a scalable hierarchical directory scheme, whereas the published literature on the I-ACOMA does not specify the details of the directory scheme; in fact, the I-ACOMA literature does not focus much on the data migration aspects of a COMA, but more on latency hiding schemes through the use of simultaneous multithreading and its implementation using embedded memory process technology. As mentioned previously, COMAs deal directly with the data migration issue as a cache line placement issue. In the DDM, a cluster of processors share an "attraction memory" (AM) where requested data

is stored; frequently requested data naturally migrates and clusters around the processors that require the data. The location of data is tracked using a hierarchical directory lookup based on point-to-point wiring, as opposed to the KSR-1 which uses a series of interlocking rings to resolve the location of data. While the point-to-point hierarchical lookup addresses some of the scalability issues of the KSR-1 interlocking rings, it still relies on a directory lookup architecture. This means that either large cache lines or a high memory overhead must be paid for storing the presence bit vectors in the cache memories. While there are mechanisms such as sparse directories [GWM90] or limited pointers [ASHH88] that can reduce this overhead, these mechanisms introduce more complexity into the system. The ADAM architecture, on the other hand, presents programmers with a virtual shared memory space and no caches. Coherence in ADAM is trivial, as there is only one location for any mutable piece of memory; hence no complexity or performance is lost to a directory cache scheme. The performance loss of not caching memory locally is gained back through three methods. The first is a simple network protocol and architecture that enables low latency remote memory requests. The second is aggressive multithreading to hide fetch latencies, in the style of HEP. [Smi82a] The third is the use of both data and thread migration mechanisms that supplant the locality of data nominally provided by directory caching schemes.

NUMA architectures make the reality of non uniform memory access an explicit architectural assumption, and typically provide automatic mechanisms to hide the latency of remote memory accesses. In the case of the Stanford DASH [CDV+94] and the SGI Origin 2000, a directory-based cache coherence protocol is employed to help enhance data locality and re-use. The amount of data that can be "migrated" locally in a ccNUMA architecture is limited by the size of the cache. Unlike the DDM COMA, the alloca-

tion, placement, and coarse migration of data is explicitly managed mostly by software; still, fine-grained data migration is provided by the caching mechanism. Because of the large overheads incurred by software page migration management, these ccNUMA machines fall into the class of coarse-grained data migration machines. On these machines, it is impractical to consider a migration system where data is dynamically and frequently moved around to reduce latency and balance loads. For example, the SGI Origin 2000 provides hardware-supported page migration through two mechanisms: per-page reference counters for profiling, and a direct memory access (DMA) style block transfer mechanism to accelerate page copying. The time required to copy a page of memory is under 30 μs; however, the time required to invalidate and update the TLBs is 100 μs or more. [LL97] While a technique called "directory poisoning" is provided that allows the TLB update to overlap with the page copy process, the performance of page copying is still less than desired.

4.2.2 Soft Migration Mechanisms

A number of innovative, high performance mechanisms have been proposed for the efficient migration of threads for load balancing within more conventional architectures.

TAM [CSS+91] (also referred to as Active Threads in [WGQH98]) and its follow-on, Active Messages [vCGS92], proposes an efficient mechanism for interprocessor communication using continuations. It significantly differentiates itself from the J-Machine [NWD93], Monsoon and *T [PBB93], all message-driven machines, by the fact that Active Messages is a purely software-approach to achieving high performance. [vCGS92] claims that pure message-driven hardware implementations are crippled by the limited number of registers available per hardware context, whereas a software emulated implementation could leverage the rich architecture of a conventional

processor. It also differentiates itself from other message passing systems by operating entirely in user space, so as to cut out kernel overheads, and by allowing concurrent message transmission and computation through non-blocking operations. Active Messages demonstrated a performance of 11 μs (21 instructions) to send a message and 15 μs (34 instructions) to receive a message on an nCUBE/2. On a CM-5, performance is 1.6 μs to send a single-packet (address + 16 message bytes) and 1.7 μs for receiver dispatch. Significantly, Active Messages is not a thread migration mechanism; rather, it is a method for compile-time integration of fast message passing mechanisms, similar in nature to Remote Procedure Calls (RPCs). Thus, Active Messages does not address how to deal with spatially nonuniform memory or situations where it is difficult to statically analyze the optimal pattern of thread creation and messaging.

Computation Migration is a term coined by [HWW93]. Computation migration is similar to thread migration, but lighter in weight (but not as light weight as TAM threads). This paper goes into depth about the difference between RPC, data migration and computation migration. A prototype system based on PROTEUS (an object oriented language) with explicit programmer annotation for migration opportunity points was used to evaluate the viability of computation migration. The implementation was tested on a counting network and a b-tree benchmark. The performance of hardware supported Computation Migration is favorable when compared to hardware shared memory and hardware supported RPC. Computation Migration is particularly good under high contention situations. Perhaps the most interesting contribution of [HWW93] with respect to this work is a detailed breakdown of where time is spent in the migration protocol. Of the 651 cycles required to migrate computation, 74% is consumed by "message overhead", *i.e.*, moving memory around, scheduling, marshaling data, creating threads, and dealing with

procedure linkages; only 3% is consumed in network transit and the remaining 23% is consumed by what appears to be user code annotations. User code annotations are required under this scheme as migration is explicitly managed by the user. Note that Active Threads [WGQH98], a slower migration scheme, is used as the comparison point for my work over Computation Migration because these static annotations restrict the utility and concurrency benefits of Computation Migration. Even so, my thread migration mechanism performs about an order of magnitude faster, cycle-for-cycle, than the Computation Migration scheme. [Hsi95] describes an extension to the work where dynamic migration is implemented using a system called MCRL. Migration decisions are based on a pair of simple heuristics based on the frequency of reads and writes. Benchmarks run on the MIT Alewife system [ABC+95] indicate that computation migration can be used in combination with data migration in situations where shared memory writes are common to improve performance. ADAM expands upon this work by creating a hardware mechanism for lowering the overhead of thread and data migration and thus enabling efficient fine-grained migration.

Active Threads [WGQH98] is a paper that describes a thread migration mechanism that employs a user-space threading scheme similar in spirit to Cilk [Joe96], Filaments [LFA96], and Multipol [WCD+95]. Active Threads stripe processor node addresses across a large virtual memory space to avoid having to update thread pointers upon thread migration. Without special hardware support, Active Threads achieves a 17 μs one-way latency for a 5 word message. A bulk transfer of 1 kbyte takes 560 μs, constrained by the host I/O bandwidth. A thread with a null stack can be migrated in 150 μs; on a Sparc v8 architecture processor using gcc 2.7.1, a null thread stack is 112 bytes. A 2 kbyte stack takes 1.1 ms to migrate. These tests were run on a cluster of 50 MHz Sparcstation 10s with Myrinet. The paper compares this thread migra-

tion mechanism against schemes such as Ariadne, Millipede, and PM^2; these other schemes have a performance on the order of 10 ms for basic migration operations. Finally, *super-linear* speedup is demonstrated for locality-guided migration on a simple multithreaded `grep` application searching across a distributed disk array. Average thread lifetimes in this benchmark are on the order of 5-10 ms. The ADAM architecture adopts Active Thread's use of a node-striped address space but also enhances performance by providing a hardware mechanism to accelerate migration and by providing temporal bi-directional pointers to perform lazy pointer updates.

DEMOS/MP [PM83] is an operating system that implements an efficient thread migration mechanism. The thread migration mechanism described in DEMOS/MP is very similar to that used in ADAM, but implemented entirely in software. DEMOS/MP processes consist of program state, link tables, message queues and "other state" (presumably heap state). Inter-process communication occurs through OS allocated and administered links that are recorded in the link tables. This use of explicitly managed inter-process communication links enables DEMOS/MP's efficient process migration mechanism. When a process wishes to migrate, it is halted, space is allocated on the remote node, and the process is moved. Messages accumulated during migration are forwarded on to the new process location, and there is a mechanism for updating sender link tables to reflect the new process location. There is little mention of performance and a dearth of comparison benchmarks in [PM83], but the paper does mention that a null thread–one with no program or data information–has a size of 850 bytes total. The paper also mentions that in non-trivial processes, the size of the data and program information regions are much larger than the size of a null thread. Thus, one might safely assume that the overhead of migration is fairly high in DEMOS/MP, as its processes are roughly equivalent in structure to those found on modern

UNIX systems. The ADAM architecture improves on the DEMOS/MP migration mechanism by using a lightweight thread representation that is faster to move, and by providing an architecture that enables hardware support for interprocess communication mechanisms. Thread migration under ADAM also does not require the movement of the heap state or traversing OS-based memory allocation tables. ADAM's architecture and migration mechanism also enables data migration in addition to thread migration.

4.2.3 Programming Environments and On-Line Migration Algorithms

A hardware mechanism's design is incomplete without thought for the programming environment or algorithms required to harness the power of the mechanism.

Emerald [JLHB88] is the seminal work in object migration systems. The only other works cited by this work are the distributed Smalltalk implementation, Argus, and Eden; one might also count Hydra and Clouds (object-based operating systems) as previous work. Emerald is a system design, and embodies a language and an implementation. The language has a type system that allows the programmer to give hints to the compiler. It also provides for migration, allocation, and affinity hints in the language. Emerald is also garbage collected. The language uses a global unique name space. Objects may have processes attached to them, or they may be direct data; the decision to attach a process to an object is made by the compiler. Emerald has a strong focus on maintaining good local-invocation performance despite providing the ability to migrate objects. Forwarding pointers with timestamps are used as the method for migrating objects quickly without having to drag the universe along with a moving object. The decision of what parts of an object to move is made by the runtime and compiler; small pieces of data get

moved at migration time; larger pieces require more thought. Emerald also provides a global object lookup facility. One problem with Emerald is the handling of processor registers: an incoherency can result in processor register state due to the way activation records are moved. In the paper, Emerald was demonstrated to have good performance over a non-migrating implementation of a distributed mail-handling application. Finally, the paper provides a good summary of the benefits of migration: load sharing, communications performance, availability, reconfiguration, and the easy utilization of special capabilities.

Ciupke, Kottman, and Walter [CKW96] proposes a framework for enabling programmer-guided object migration in their paper titled "Object Migration in Non-Monolithic Distributed Applications". The paper posits that an object-oriented model is a natural match for a migratory framework, since objects naturally define a locality of data and the methods that can modify it. The paper suggests that the basic linguistic primitives required to guide migration are fixing operations, movement operations, and attachment notations. The paper also assumes that all high-level migration decisions are coded by "reasonable users". The language primitives are tested within an abstracted simulation environment that makes assumptions such as a fully-connected network. The simulations indicate that dumb migration (basic user-coded migration) yield roughly the same performance increase as profile-based migration. The results also indicates that migration can be detrimental in situations where migration policies are coded with only one component in mind. In particular, performance is degraded in the hot-spot case, and in the case that the work set of objects are tightly associated but migrated as individual entities.

"Profiling Based Task Migration" by Baxter and Patel [BP92] focuses on migration for load-balancing only, and has a very specific, limited data set.

However, it demonstrates that for this particular example, a migration algorithm acting on local knowledge only can achieve within 5% the performance of a global knowledge solution.

Kalogeraki, Melliar-Smith, and Moser [KMSM01] discusses dynamic algorithms for distributed object migration in their paper titled "Dynamic Migration Algorithms for Distributed Object Systems". This work considers systems with only 8 nodes and about 5 objects per node. Object state transferrance is hindered by the movement of OS/kernel state under the ORB [Inc01] distributed object architecture; object scheduling happens in milliseconds, profiling over seconds, and migration over tens of seconds. The test system is 167 MHz ULTRA Sparc using VisiBroker ORB 3.3, and the interconnect is 100 MBit/s ethernet. The focus of the paper is the use of migration to satisfy real-time system constraints, so the results demonstrate that "laxity" can be preserved through migration. Thus, the relevant section of this paper to this thesis are its dynamic migration algorithms. The paper presents "cooling" (load balancing) and "hot spot" (latency reduction) algorithms, evaluated independently. The algorithms correspond to the intuition brought by their names, and the paper demonstrates that these algorithms can be used to successfully balance a task within a distributed system.

The Object Request Broker (ORB) [Inc01] system used by [KMSM01] is described in a 1000+ page document. ORB is an open architecture and specification for defining objects that can be shared, interoperated, and invoked under one huge common umbrella. As noted previously, the overhead incurred by the ORB system places it in a different league of migration systems when compared to this thesis; however, the standard itself addresses a number of interesting programming issues that are beyond the scope of this thesis.

In the context of real time distributed object systems, [HS94b] uses Bayesian

analysis and queuing theory to determine if a task should be migrated to a destination node given a set of real-time constraints and some estimates about the task's execution time and laxity. The article references a prior work [HS94a] which describes how to estimate the load state of a remote node given outdated information. This article focuses primarily on ensuring that decisions to transfer work to another node are done in such a manner that future task arrivals are also considered. This prevents the situation of everyone sending their tasks to the one unloaded node in the network just because the outdated load information looked good at the time of migration initiation. This article also introduced the idea of "buddies" that are physically co-located for restricting the range of broadcast state information and attempts to reduce the amount of communication required to maintain other-node state.

This article is a good example of a formal analysis of data migration in a complex system using statistical analysis. Other methods for analyzing the system could be through control systems theory (feedback systems) or through on-line competitive analysis. A significant difference of this article from the work I am concerned with is that this work investigates real-time systems, whereas my work is simply interested in optimal performance (minimum execution time as opposed to guaranteed time of execution).

Hall, *et al.* [HHK+01] presents a theoretical paper on data migration in the context of load balancing and optimizing a storage system. A fully connected, bidirectional network is assumed, with objects all of the same size. Even with these assumptions, the problem of determining an optimal plan for data migration is declared to be NP-complete. The problem is also NP-complete for just two nodes directly connected with objects of variable size, given that only one object can move at any time and that space is very limited on each node. The paper claim that this problem is equivalent to edge-coloring for the unconstrained space problem, and very similar in solutions

bounds to the edge coloring problem for constrained space problems. The good news is that heuristics and poly-time algorithms are available that can solve the problem to near-optimality. [BEY98] is a survey work on competitive analysis and on-line algorithms that describes some of the algorithms that can be applied to data migration and load balancing problems. I base much of the formal analysis in my thesis on the contents of [BEY98].

4.3 Migration Mechanism Implementation

The Q-Machine is an implementation of the ADAM abstract architecture. The Q-Machine leverages ADAM's architectural features to enable fast, low-overhead migration mechanisms. This mechanism reduces the latency and bandwidth cost of migrating lightweight data and threads to that of an L2 cache fill on a Pentium 4 processor in a RAMBUS based system. The estimated system latency of an L2 cache fill is about 175 ns (which is 140 800 MHz Direct-RAMBUS cycles) [CJDM01], and the size of an L2 cache line is 128 bytes [HSU+01]. Note that the Pentium 4 processor's L2 cache is sectored into 64-byte halves, but according to [HSU+01], L2 cache fills "typically" fetch data for both sectors. More information on the performance of the migration mechanism can be found in section 6.

The ancillary details of the Q-Machine implementation are presented in chapter 5; for now, I will focus solely on the implementation details relevant to data and thread migration. Also, when reading this section, it is assumed that the reader is familiar with the ADAM architecture specification (chapter 3 and appendix B).

The heart of the migration mechanism is the tagged capability architecture of the ADAM, and the use of queue maps for inter-process communication. These two hardware enforced disciplines drastically reduce the amount

of bookkeeping and special-purpose hardware required to implement an efficient migration mechanism. Capabilities encode their base and bound information within their tags, so the boundaries of migrated data are explicit. In addition, capabilities in this architecture feature an "increment-only" bit that allow portions of the beginning of the capability to be safely reserved for overhead functions such as forwarding pointers and statistics bookkeeping. Also, thread state has a one-to-one mapping with a capability (the thread's context ID) in memory due to the named-state queue file implementation (see appendix C for queue file implementation details). This feature allows thread migration to share almost all of the mechanisms of data migration; the primary difference is that thread migration requires additional locking and synchronization with the physical queue file. The use of queue maps for interprocess communication is important because it enables simple mechanisms for synchronizing, redirecting and updating inter-thread communications requests during and after a thread migration event.

4.3.1 Remote Memory Access Mechanism

I will now introduce the remote memory access mechanism used in the Q-Machine implementation. The remote memory access mechanism is an important component of the migration mechanism. Recall that the address space of ADAM is structured so that the processor node ID is the highest address bits; also, by convention, processor nodes occupy the even route addresses, and memory nodes occupy the odd route addresses. This allows processors and memory to be paired off into "preferred" pairs by the existence of a reliable, in-order delivery cut-through network path between preferred pairs. A local memory access is thus defined as a memory access where the node ID of the access capability is equal to the node ID of the preferred memory node. Local memory accesses are always serviced by the preferred mem-

ory node, and local memory allocation requests allocate data in the preferred memory node. The performance of accessing data in the preferred memory node is similar to that of an L3 cache access time on a contemporary processor; please see section 6 for specific numbers.

Semantically, a preferred memory node is the target of all `MML`, `MMS` and `EXCH` queue mappings, regardless of the access capability used to initialize the mapping. Thus, all remote requests are also routed from a processor node to the preferred memory node. When a remote request is initialized, the local virtual memory handler allocates local "shadow" pages for the remote capability. Shadow pages serve two functions: first, they provide a method for storing the remote memory's data locater pointer; second, they provide the infrastructure for caching immutable data. Shadow pages should never displace local memory pages when local memory is scarce. Hence, most of the shadow pages are not swapped into core or initialized when they are first allocated. The only exception is the first page. The first memory location of the first shadow page is the data locater pointer. This data locater pointer is initialized with the remote access capability. Note that the rest of the first shadow page's space is marked as all invalid and all non-primary. Figure 4.1 illustrates the format of a remote capability in shadow space.

Figure 4.2 overviews the system level view of resolving a remote memory request. Remote requests are easily detected when a memory-mapped queue is initialized with its access capability: if the node ID of the access capability is not equal to the memory node's ID, it must be a remote request. This remote request status is noted in the memory node's access table tags (for more information on the memory node access table, please see section 5.3.2).

All requests from a processor node to a preferred memory node use the format of a transport packet without the physical layer route header and checksums. More information on the transport protocol used in the Q-Machine can

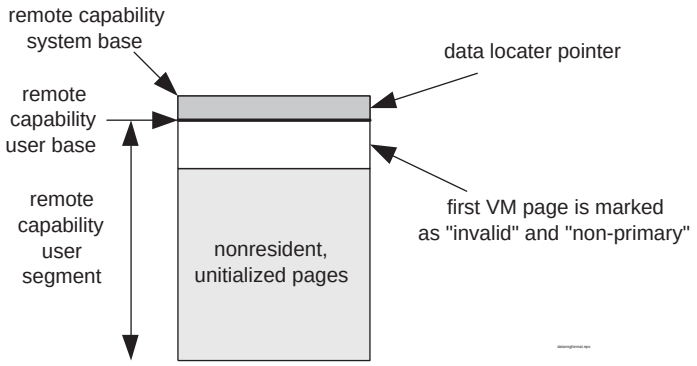


Figure 4.1: Format of a remote memory capability's shadow space in local virtual memory space.

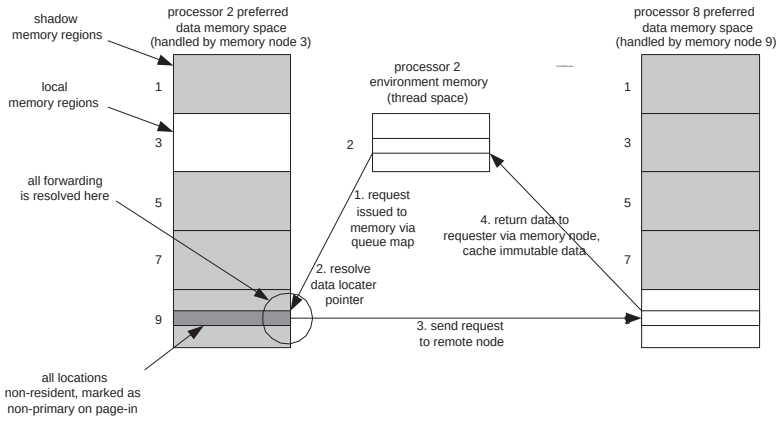


Figure 4.2: System level view of resolving remote memory requests.

be found in appendix C.2. These transport packets contain all the state required to resolve the return address of the requester; thus, when forwarding a memory request, the memory node simply encapsulates the processor's original request packet in forwarding headers and sends the encapsulated packet on to the remote memory node. Please see figure 4.3 for a more detailed illustration of how local and remote exchange (EXCH) mappings are handled. The EXCH operation was chosen for illustrative purposes because it combines both a load and a store operation. A load operation uses exactly the load-half of the EXCH protocol, and a store operation uses the store-half of the EXCH protocol, plus a store-acknowledge packet so that writes can be guaranteed to complete in program order.

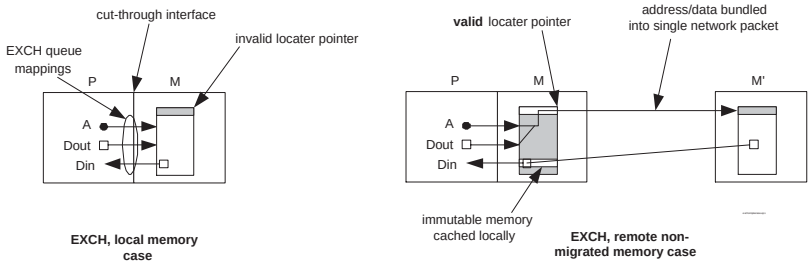


Figure 4.3: Details of handling remote and local EXCH requests.

Note that for compatibility with the migration mechanism to be outlined in the next section, all memory accesses, even stores, must check the valid and primary tag bits. If the existing value is invalid and the non-primary bit is set (as is the case when data has been migrated out), then the access table must be updated to forward future requests, and the current request must also be forwarded to the remote request queue. The overhead of tag checks on all requests, including stores, can be mitigated if dedicated hardware is provided in the memory implementation.

4.3.2 Migration Mechanism

The remote memory access primitives described in the previous section enable the streamlined implementation of migration mechanisms. The migration mechanism recognizes only two commands, migrate data ($M_D(C_{local}, ID_{dest})$ and migrate thread ($M_T(C_{local}, ID_{dest})$). The arguments to these commands are the source capability of the data or thread to migrate, and the destination processor ID. Other commands that could be implemented include partial migration commands and copy immutable data commands.

4.3.3 Data Migration

The data migration mechanism implemented in the Q-Machine relies on the following assumptions and invariants.

Invariant: *The user only sees one global unique name for each capability, and this name never changes.* This is enforced by the basic data locater pointer at the top of every capability. This data locater allows the actual data to move freely without having to concurrently modify thread state.

Assumption: *There is at most one outgoing migration process per memory node at any given time.* This assumption simplifies the hardware requirements for freezing and synchronizing access requests to a piece of data in flight.

Invariant: *The relative order of requests to any given capability is preserved before, during, and after migration.* This is important in maintaining consistency in the memory model.

Assumption: *The relative order of requests between the migrating and the non-migrating capabilities is not important.* This is a general assumption of the architecture, but it is restated here for clarity. It is the requestor's responsibility to ensure, for example, that stores to one location complete

before loads to the same location. In the Q-Machine implementation, only one pending request is allowed per thread per unique memory location; a higher-performance solution may use store buffers with associative lookup to alleviate this bottleneck, so long as it does not cause problems with the next assumption.

Assumption: *There is only one pending memory request per thread per unique memory location in the network at any given time.* This rather restrictive assumption is required because requests to a migrating capability are delayed for the duration of a migration event; in fact, in the case that data is migrating across a routing bottleneck, requests issued after migration will arrive before any pending requests issued before migration. It is possible to relax this assumption with extra bookkeeping in the migration mechanism and pointer update protocol, but this kind of performance optimization complexity is eschewed in my research prototype. Note that local requests have less restrictive requirements because the cut-through interface has stronger request ordering guarantees than the external network interface.

Invariant: *There is at most one primary copy of a capability within the system at any time.* The primary copy of a capability is the copy that is allowed to respond to load requests for mutable data or any store or exchange request. A capability is primary when the primary tag bits are set on all the data within the capability's segment.

Invariant: *When there are zero primary copies of a capability within the system, no requests to the capability are serviced.* In other words, data in flight cannot be modified or read.

Assumption: *A capability to be migrated starts out local.* A memory node cannot manage the migration of capabilities that are not local; if this must happen, the local memory node should send a message to the remote memory node to request a migration.

Performance Tip: *It is helpful to have a hardware mechanism for clearing or setting the primary bits on large blocks of memory.* This is a frequent operation performed by the migration mechanism that scales poorly with the size of the capability segment. One implementation approach could be to interleave the primary-bit clearing operation with the readout of the data during the copy phase of migration.

This is the procedure for migrating a capability.

- A migration request is issued of the format $M_D(C_{local}, ID_{dest})$
- A request to allocate a capability C_{remote} of suitable size is issued to the receiving memory node.
- Requests to C_{local} are serviced until C_{remote} is returned to the source node.
- All incoming network requests to C_{local} are frozen using the mechanism diagrammed in figure 4.4. Note that outgoing data may continue to be sent and resent by the idempotent sequenced transport protocol outlined in section C.2.
- C_{local} is copied to C_{remote}
- The contents of C_{local} are marked as invalid and non-primary, except for immutable data.
- The data locater entry of C_{local} is changed from invalid to C_{remote}.
- Outgoing data in-flight prior to the freezing of C_{local} must all be acknowledged in accordance with the sequenced idempotent network protocol before continuing to the next step.
- Requests to C_{local} are unfrozen and re-scheduled. These requests are now handled by the existing remote memory access infrastructure.
- Eventually, after all pointers to C_{local} have been updated, the garbage collection mechanism de-allocates C_{local}.

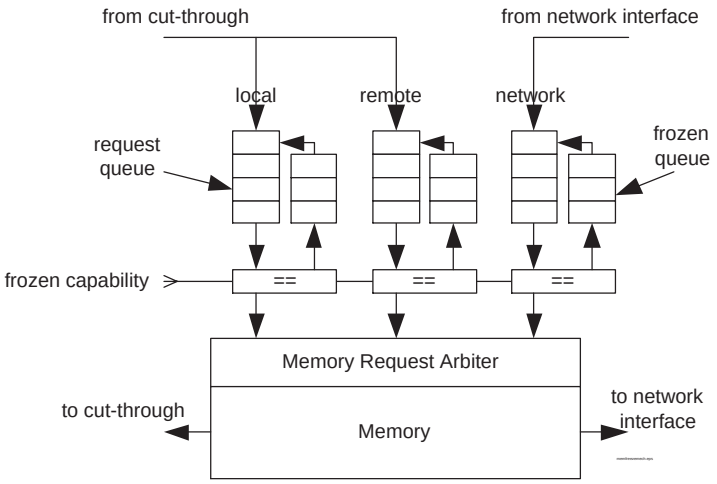


Figure 4.4: Mechanism for temporarily freezing memory requests.

In addition to the migration mechanism, a mechanism is required to update *incoming* data locater pointers, or else every memory request will eventually have to traverse a chain of data locater pointers. One method for performing pointer updates is to sweep through memory and resolve all data locater pointers to their primary locations. This method is prohibitively expensive and slow. A better solution is to employ bi-directional data locater pointers, and to send update messages along the reverse paths every time a piece of data is migrated. However, bi-directional pointers have the drawback of needing to maintain an arbitrarily large list of reverse pointers. The reverse pointer update also jams the outgoing network ports of the memory node if the reverse pointer list is large. In order to counter these faults, I use a mechanism I call temporally bi-directional pointers.

Temporally bi-directional pointers can be thought of as lazily evaluated bi-directional pointers. Whenever a request is issued to a migrated capability, the response is a pointer update message. This pointer update message

contains the body of the original request so that the requester does not need to keep track of outstanding request state. Please refer to figure 4.5. While the temporally bi-directional pointers mechanism wastes one trip across the network per update when compared to eagerly updated bi-directional pointers, temporally bidirectional pointers have many advantages: they require constant space to implement; reverse pointers that are inactive consume no resources; update requests are spread out over time so the effective latency is lower due to less queuing of requests; and, if a data block was migrated across a bottleneck, the bottleneck is not aggravated by a deluge of update messages.

4.3.4 Thread Migration

The thread migration mechanism implemented in the Q-Machine relies on the following assumptions and invariants.

Invariant: *The user only sees one global unique name for each thread, and this name never changes.* This is enforced by the data locater pointer at the top of every capability, including thread capabilities. This data locater allows the actual data to move freely without having to concurrently modify thread state.

Invariant: *All inter-thread operations are write-only.* This comes for free with ADAM's "push" model of inter-thread communications. In other words, only outgoing queue maps are allowed; a local queue cannot request "read" data out of a remote queue.

Assumption: *There is at most one outgoing migration process per processor node at any given time.* This assumption simplifies the hardware requirements for freezing and synchronizing requests to a thread in flight.

Invariant: *The relative order of requests to any given thread is preserved before, during, and after migration.* This is important for maintaining consis-

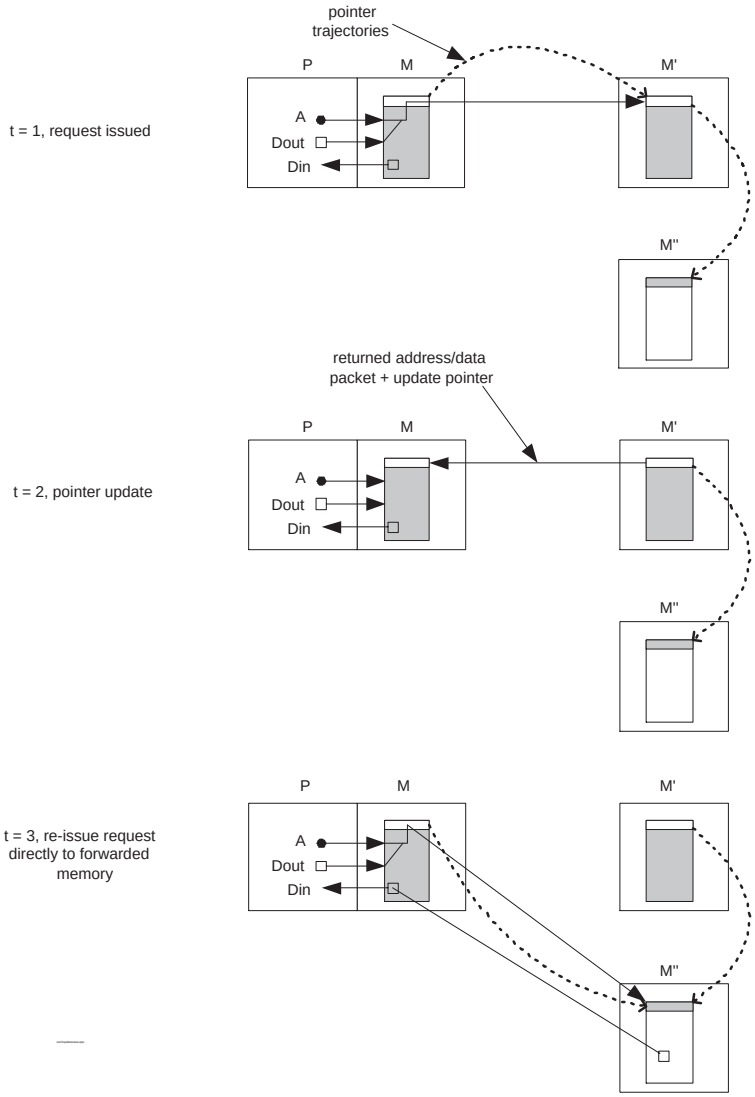


Figure 4.5: Handling of a migrated EXCH request with temporally bi-directional pointers.

tent data ordering in mapped queues.

Assumption: *The relative order of requests between migrating and non-migrating threads is not important.* This is a general assumption of the architecture, but it is restated here for clarity.

Invariant: *There is at most one primary copy of a thread within the system at any time.* The primary copy of a thread is the copy that is schedulable and is a valid target for incoming data from mapped queues. A thread is primary when the primary bit is set on the context ID. Recall that the context ID is also the capability for the backing store of the thread.

Invariant: *When there are zero active copies of a thread within the system, no requests to the thread are serviced.* In other words, a thread in flight cannot be modified or read.

Assumption: *The thread to be migrated starts out local.* A processor node cannot manage the migration of threads that are not local; if this must happen, the local processor node should send a migrate request message to the remote processor node.

Performance Tip: *The hardware should keep track of queues that have been created, in addition to exactly which queues have memory maps, source maps, and drop maps applied to them.* With this information, queues that have not been created (*i.e.*, never referenced or otherwise empty) can consume zero overhead during migration. The named state queue file implementation of the Q-Machine provides all of this bookkeeping information for free.

The Q-Machine implements two procedures for migrating threads. One is used when the thread is determined to be "lightweight", *i.e.*, it has few memory mappings, few created queues, and little other state associated with it. The other is used when a thread is determined to have a large amount of state and may cause loading problems on the network and the receiver; this is referred to as a "heavyweight" thread. The primary difference between the

protocols is the timing of the remote thread allocation versus the arrival of the thread state. A heavyweight thread migration issues an allocate request before sending the thread data; this allows migration preparations to occur in parallel with the servicing of the allocation request. A lightweight thread migration piggy-backs the thread state along with the remote thread allocation request; this optimization reduces the time required for a lightweight thread to migrate.

The following is the procedure for migrating a lightweight thread; the responsibility for handling this migration is split between the sender and the receiver.

Sender's Procedure for a Lightweight Thread Migration:

- A migration request is issued in the format $M_T(C_{local}, ID_{dest})$
- Thread C_{local} is removed from the local pool of threads. This includes the processor node work queue and any internal state maintained by the thread scheduler.
- All incoming requests to C_{local} are frozen using a mechanism similar to that in figure 4.4.
- All of C_{local}'s state is flushed from the physical queue file into environment memory.
- All pending memory requests for C_{local} must complete or at least be acknowledged before executing the next step.
- All pending outgoing requests of C_{local} in the transport layer must be acknowledged before continuing on to the next step.
- C_{local}'s thread state is migrated to node ID_{dest}.
- C_{local} is set to non-primary.
- Once a "migration successful" message has been received from ID_{dest}, the pending requests to C_{local} can be unfrozen and re-scheduled. The pointer update mechanism, discussed later, handles these requests.

- Eventually, after all pointers to C_{local} have been updated, the garbage collection mechanism de-allocates C_{local}.

Receiver's Procedure for a Lightweight Thread Migration:

- An incoming migration packet is received containing C_{local}'s thread state.
- C_{dest} is allocated, and C_{local}'s thread state is copied into C_{dest}.
- C_{local}'s memory mapped queues are reconstructed for C_{dest} to the receiver's preferred memory node.
- C_{dest} is immediately placed into the receiver's runnable thread pool.
- A "migration successful" token is sent to the sender.

The following is the procedure for migrating a heavyweight thread. Again, the burden of the protocol is shared between the sender and the receiver.

Sender's Procedure for a Heavyweight Thread Migration:

- A migration request is issued in the format $M_T(C_{local}, ID_{dest})$
- A request to allocate a capability C_{remote} of suitable size is issued to the receiving processor node (node ID_{dest}).
- Thread C_{local} is removed from the local pool of threads. This includes the processor node work queue and any internal state maintained by the thread scheduler.
- All incoming requests to C_{local} are frozen using a mechanism similar to that in figure 4.4.
- All of C_{local}'s state is flushed from the physical queue file into environment memory.
- All pending memory requests for C_{local} must complete or at least be acknowledged before executing the next step.
- All pending outgoing requests for C_{local} in the transport layer must be acknowledged before continuing on to the next step.
- C_{remote} must be received before continuing on to the next step.

- C_{local}'s thread state is migrated node ID_{dest}.
- C_{local} is set to non-primary.
- Once a "migration successful" token has been received from node ID_{dest}, the pending requests to C_{local} can be unfrozen and re-scheduled. The pointer update mechanism, discussed later, handles these requests.
- Eventually, after all pointers to C_{local} have been updated, the garbage collection mechanism de-allocates C_{local}.

Receiver's Procedure for a Lightweight Thread Migration:

- An incoming allocate thread request is received; space is allocated and C_{remote} is returned to the sender.
- C_{local}'s thread state is received and copied into C_{remote}.
- C_{local}'s memory mapped queues are reconstructed for C_{dest} to the receiver's preferred memory node.
- C_{dest} is placed into the runnable thread pool.
- A "migration successful" token is sent to the sender.

Queue mapping pointer updates are handled in a slightly different manner than the data locater pointer updates for data migrations, because multiple requests are allowed to be outstanding to a single queue during thread migration. A "transmission line" protocol is used in this case. The name was chosen to reflect the similarity of this situation to the propagation and reflection of waves in a series-terminated transmission line. Please see figure 4.6. This protocol relies on one additional assumption.

Assumption: *All messages between a sending and receiving thread are guaranteed to be delivered and processed in-order with respect to the message sequence generated by the sending thread.* This assumption is enforced by the idempotent sequenced transport protocol used in the Q-Machine implementation.

This is the transmission line protocol:

- A thread is frozen due to a migration in progress
- Incoming requests to the frozen thread are blocked, and a "forwarding pointer update" packet containing the blocked incoming requests is returned to the sender as soon as the thread has been migrated.
- Once the sender receives the "forwarding pointer update" packet, it issues a "forwarding acknowledge" message and ceases to issue any further requests to the migrated thread; meanwhile, the sender re-issues any returned requests to the new thread location
- Once the old location receives the "forwarding acknowledge" packet, it sends an "okay to unblock" packet to the sender.
- Once the sender receives the "okay to unblock" packet, the sender may issue new requests to the migrated thread.

4.4 Migration Mechanism Issues and Observations

A detailed review of the performance of the migration mechanism can be found in section 6. This section reflects on some of my general observations about the migration mechanism and its design and implementation challenges.

4.4.1 General Observations

The basic protocol for migration in a capability-based architecture with native support for forwarding pointers is simple: lock the capability, move the data, then unlock the capability. Of course, the devil is in the details. It is the difficult details that lead to the dichotomy of memory and thread capabilities.

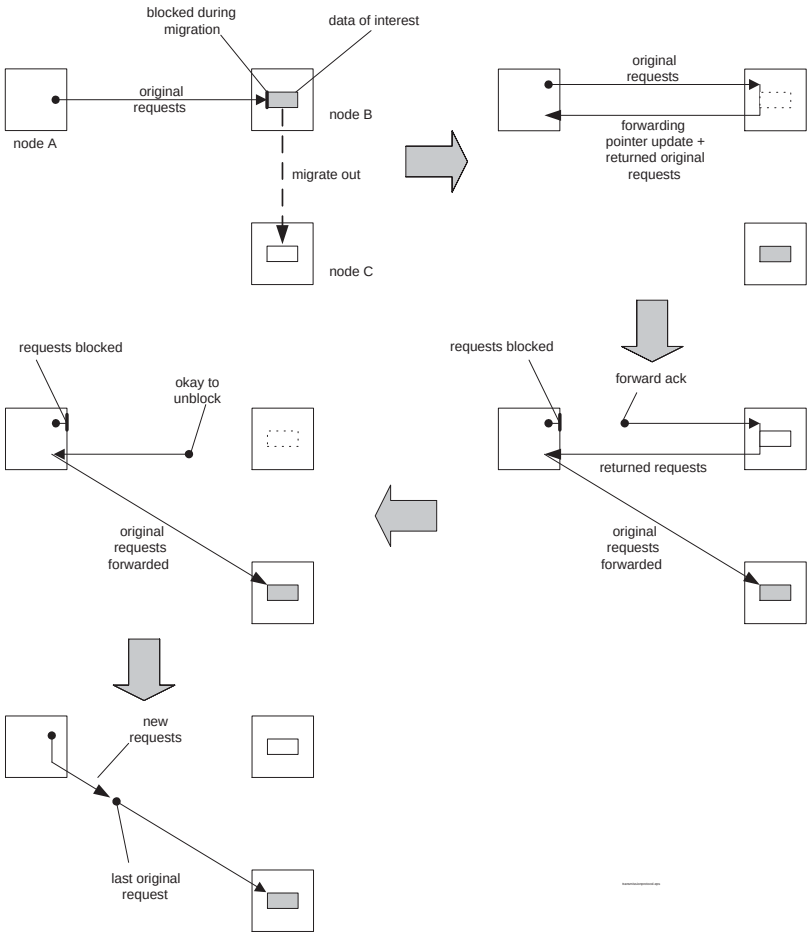


Figure 4.6: Transmission line protocol for handling forwarding pointer updates on thread-mapped communications.

In other words, while it has been pointed out that perhaps a single mechanism could be used for both threads and data, it seems that the nature of how memory and threads are used by programmers leads to a natural segregation of these structures in the machine architecture.

One basic trade-off enabled by treating data and threads separately is a simplification of the memory migration protocol. Since memory has no auxiliary state – no queue file, no scheduler entries, only one pending request per thread per location – movement of memory is much lower overhead than movement of a thread. On the other hand, because threads use strictly unidirectional communication, pointer updates can be managed across multiple outstanding operations. Multiple concurrent memory operations are more complicated because order has to be maintained between all possible store, load and exchange queues that may be dynamically mapped to a single memory location. Thus, by dividing the machine into distinct memory and processor nodes, the respective migration protocols can cull out any unnecessary assumptions or conditions specific to each situation.

In addition, a thread-only programming model may be advantageous in situations where performance hinges on having available multiple concurrent requests to memory. In a thread-only programming model, the programmer sees no memory nodes or memory mapped queues; instead, dedicated server threads handle memory requests in an abstract fashion. The ADAM can be specialized into a thread-only programming model using compiler tricks with some OS support. The compiler takes care of inserting the code necessary to spawn abstract dedicated memory servers, and the OS is responsible for coordinating the migration of the server threads along with their associated data.

4.4.2 Performance Issues

I designed the migration mechanism to be a high performance solution with latencies and bandwidth requirements comparable to L2 cache fills in a contemporary conventional processor. Results presented in section 6 show that I achieve this goal for lightweight threads and data. Of course, there are still optimizations that could be applied to the migration mechanism.

For example, partial migration of thread state could accelerate the time between a migration decision and the first arrival of a thread at its destination. My scheme locks down a thread and moves its entire contents before re-scheduling the thread. A more sophisticated scheme would keep track of the most recently and most commonly used set of data in the thread, and just send that data over in a small packet for immediate scheduling; as the receiving node initializes and schedules this thread for execution, the environment memory could be concurrently filled with the remaining thread data. Partial migration is feasible only because of the flexibility in the named-state queue file implementation used by the Q-Machine.

A partial migration scheme can also be applied to large data sets. If an extremely large capability is allocated, the capability could be sectored off into sub-blocks by representing the actual capability as a set of smaller capabilities within the large capability. The smaller sub-blocks would contain pointers to the parent capability, and vice versa; the pointer implementation would be similar to that of the data locater pointer scheme used by my remote memory access mechanism. These sub-blocks would be freely migrated around the system independently of the parent capability.

Another method for partial migration is demand-based copying of data. In this scheme, a set of reverse pointers are also required in addition to the data locater pointers. The “primary” capability is still responsible for handling all operations on the capability, but data for a load or exchange is propagated

to the primary capability only when requested. This scheme is illustrated in figure 4.7. This method could be useful if very sparse, large data structures are frequently migrated throughout the machine.

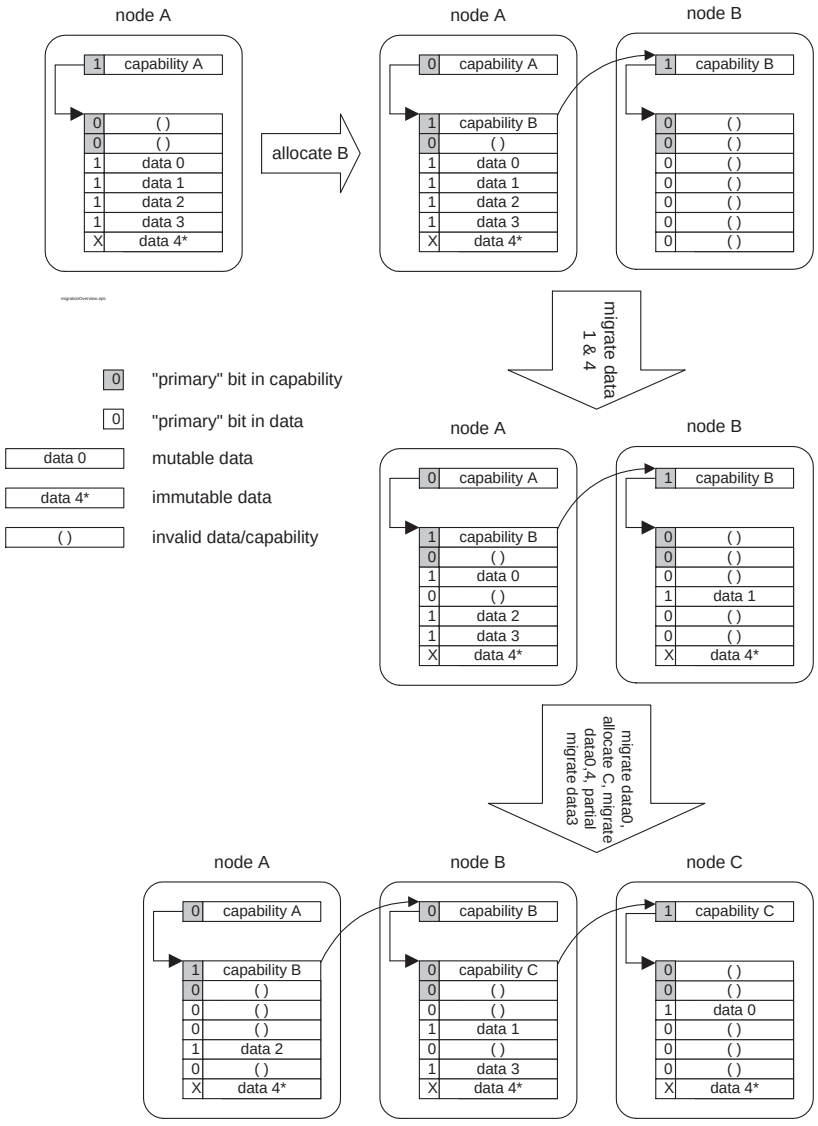
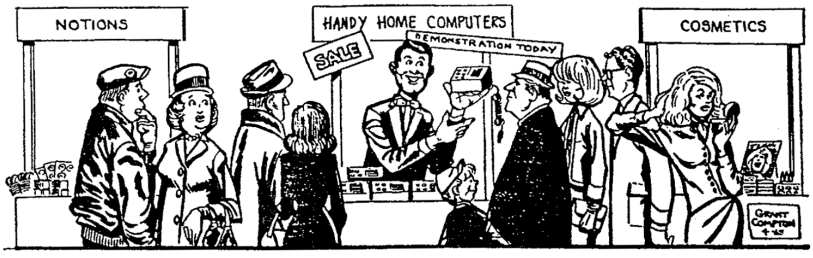


Figure 4.7: Overview of a demand-driven data propagation scheme.

Chapter 5

Implementation of the ADAM: Hardware and Simulation



From Gordon Moore's "Cramming More Components onto Integrated Circuits", April 1965

This section outlines the details of an implementation for a physical machine, called the Q-Machine, optimized to run ADAM code. A top-down approach is taken in describing the implementation. All key architectural features are

validated by a brief feasibility study in order to keep the design rooted in reality and to attempt to convince the reader that this is an implementable architecture. This section also outlines how the proposed hardware parameters are reflected in the design and implementation of a software simulator of the Q-Machine. The software simulation is bandwidth and latency-accurate, and almost cycle accurate. The first goal of the simulation is to demonstrate the feasibility of the queue-based programming model used by the ADAM and to demonstrate integration with the Couatl and People languages and toolchains. The second goal of the simulation is to demonstrate the performance of thread and data migration mechanisms described in section 4, and to provide a platform for testing various migration algorithms.

5.1 Introduction

The Q-Machine is organized as a fine-grained MIMD parallel processor tile array featuring embedded memories. A preponderance of proposed tile processor or chip-multiprocessor (CMP) architectures, many with embedded RAM of some form, have cropped up recently due to their attractive simplicity and seductive "guaranteed not to exceed" performance promises. Some of these recently proposed architectures include RAW [LBF+98], Hydra [HHS+00], IRAM [KPP+97], Sun Microsystem's MAJC, the IBM Power4, Active Pages [OCS98], Decoupled Access DRAM [VG98], Terasys [GHI94], SPACERAM [Mar00], Smart Memories [MPJ+00], and Hamal [Gro01]. A succinct article by Kunle Olukotun summarizes the essential advantages of CMPs. [ONH+96]

5.2 High-Level Organization

The address space of the Q-Machine is divided into three parts: code, environment, and data. The code space is write-once, read many and data is striped across all nodes; interaction between code space and user space is possible only through the `LDCODE` opcode. Environment and data spaces are read-many, write-many and their address spaces are local to each node. Environment space is where thread contexts are stored; thus, all interaction with environment space is implicit. Environments are "allocated" by the `SPAWN` set of opcodes, and threads are "garbage collected" as they `HALT` or are observed to no longer reference or be referenced by anything else in the system. Data space is accessed only through queue mappings in the execution unit. A memory management coprocessor is required to handle memory requests in memory space. An `ALLOCATE` opcode is provided in the instruction set as a facility to create memory, and a garbage collection mechanism is required to reclaim memory. The interaction of the `ALLOCATE` opcode with the memory management coprocessor is implementation-dependent. Figure 5.1 illustrates the high level situation that leads to this division of address spaces. Note the implementation specific options such as I/O devices and custom hardware blocks in figure 5.1. These devices can be accessed either through queue mappings set up by OS traps, or through opcodes added to the stock ADAM specifications that behave similarly to the `ALLOCATE` and `SPAWN` instructions.

5.3 Leaf Node

The basic leaf node contains two fundamental nodes: a processor node and a memory node. Each of these nodes appear identical to the primary network in terms of routing and addressing. However, a low latency cut-through

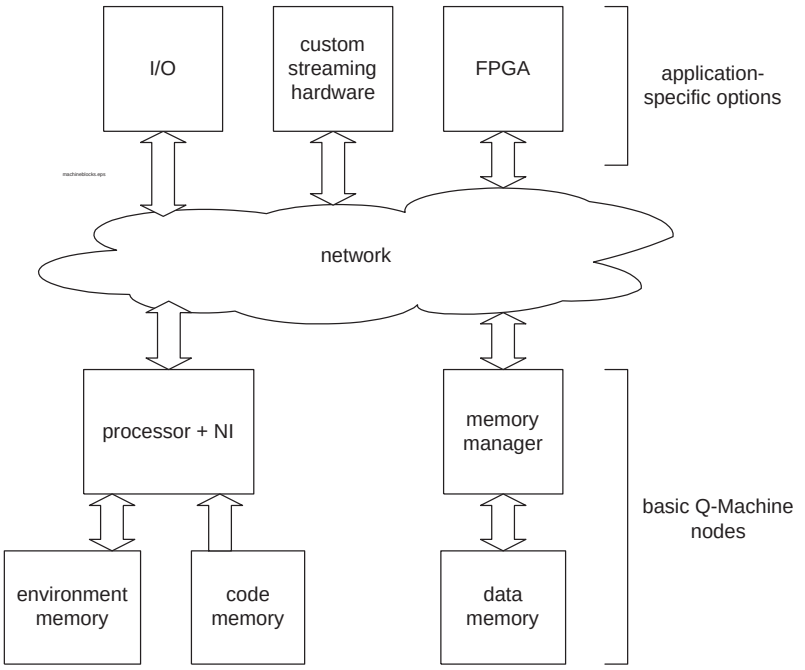


Figure 5.1: Pieces of a Q-Machine implementation. Node ID tags are uniform across the machine, so network-attached custom hardware is addressable like any processor or memory node.

path is provided between each processor-memory node pair. This path establishes the bonded memory node as the preferred location of data on which the processor node wishes to operate. The cut-through path is guaranteed to always deliver data reliably between the leaf node pair; thus, the latency associated with adding packet headers, block checksums and other bookkeeping incurred by a reliable-delivery transport protocol can be avoided. There must be sufficient bandwidth and ports available to make the probability of either the processor or memory node becoming saturated by cut-through traffic negligible. A simple way to guarantee this assumption is to partition the design so that there is a dedicated port for cut-through traffic, separate from ports for dealing with inter-node traffic. Partitioning in this manner runs the risk of dedicating excess resources to an underutilized cut-through port; however the job of the migration manager and scheduler is to try and structure the distribution of data and computation so that as much locality is exploited as possible. A block diagram of the unit leaf node implementation can be found at figure 5.2.

5.3.1 Processor Node

The processor node consists of five major sub-components: an execution unit, a scheduler, a network interface, an environment cache, and an instruction cache. An overview of the processor node organization can be found in figure 5.3.

The scheduler and the execution unit interact via a work-window path and a retired thread path. The work-window is a small buffer of scheduled threads to run. Each scheduled thread is bundled with an instruction cache line that is pre-fetched as the thread waits in the work-window. The ADAM System Simulator implements a work-window that is eight threads deep. The execution unit maintains a pointer that rotates through the work-window whenever

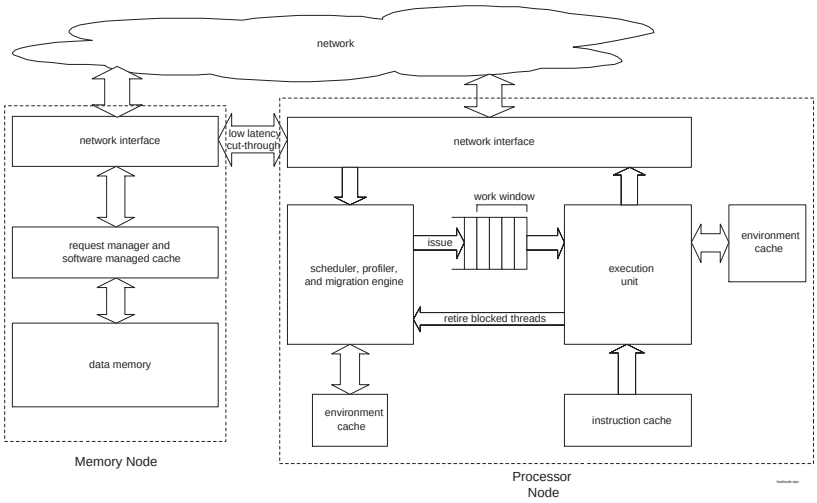


Figure 5.2: High level block diagram of a leaf node.

a thread blocks. The execution unit also maintains a run-length count of each item in the work window and forces an item to swap out when the maximum run length is reached, so as to prevent the starvation of other threads. The maximum run count is programmable at run-time.

When a thread blocks, it may be removed from the work window and sent back to the scheduler with a tag indicating on which piece of data the thread blocked. The method for determining when a blocked thread should be retired is not hard-wired.

Processor Core

The processor core itself looks similar to a classic RISC architecture. Operands are fetched from a physical queue file (PQF) and sent directly to an arithmetic unit (EXEC); results are written back, for the most part, into the PQF. The PQF is implemented in a manner similar to a named-state register file (NSRF) [ND95]. Thus, the PQF can be thought of as a cache for thread state. The PQF is fully associative: any line in the PQF can be mapped to any

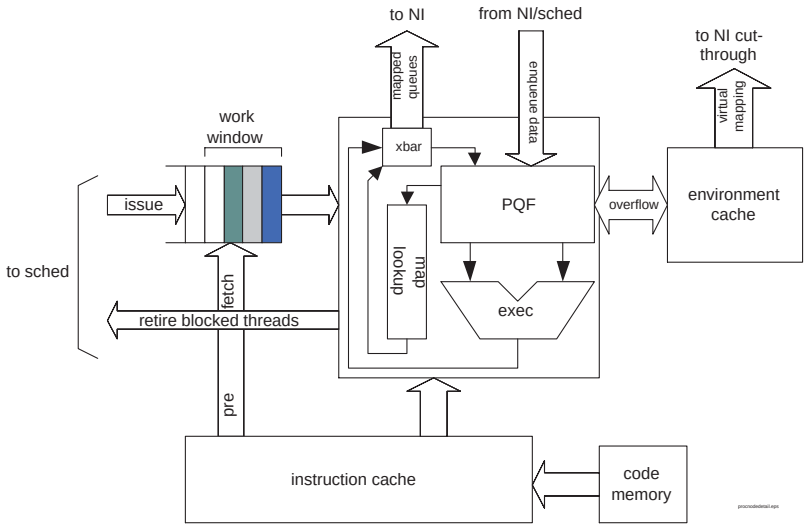


Figure 5.3: Detail of a processor node.

queue in any thread context. A contiguous region of the data memory space is dedicated to "environment memory" so that lines in the PQF have a simple one-to-one mapping with addresses in memory. This environment memory is guaranteed to be node-local via a contract with the migration manager (see Section 4 for more details). One distinguishing feature of the PQF is that it has an auto-spill feature, such that when the PQF exceeds a certain threshold of fullness, lines are retired **only** when there is available bandwidth to the environment cache. The ADAM System Simulator implements a PQF with 128 lines and an auto-spill threshold of 124 lines. Please see Appendix C for more notes on the implementation of the PQF.

Inter-thread communication occurs via the "push" model only. A push model means that a thread can only generate inter-thread traffic that targets another node; it cannot "pull" data from another node by placing an inter-thread mapping on a read port. By forcing thread communication to happen

only on data writes, the queue mapping lookup can occur in parallel with the result computation. Thus, critical path overhead is kept to a minimum for inter-thread communication. Data that is destined for a thread context located on the local processor node is immediately looped back into the local scheduler which performs some bookkeeping and then quickly forwards the data directly into the PQF.

An important observation is that when the working set of contexts have a footprint that fits within the PQF, and there is little inter-thread communication, the execution core datapath looks almost exactly like that of a standard RISC processor. This simplicity of the critical path enables the implementer to more easily achieve high clock rates in the execution core and in turn yield high performance on single-threaded code. Also note that the execution core can be easily extended to a super-scalar out-of-order issue implementation: the queue structure of the register file gives some amount of register renaming for free, and the empty bits on the PQF simplify the implementation of out-of-order dispatch.

Scheduler

The use of fine-grained multithreading to hide latency has been seen before in the Tera/MTA [AKK+95], HEP [Smi82a], M-Machine [FKD+95] and *T [PBB93], among others. The scheduling algorithm implemented in the simulator for this work is a derivative of that used in [NWD93], and takes after the general scheduling algorithm described in the introduction to this section. Threads are divided into two pools, a runnable pool and a stalled pool. The runnable pool is executed in a round-robin fashion with a thread pre-emption timeout to guarantee some fairness. Threads that block on a data availability stall are retired to the stalled pool; threads that block on a structural stall (such as a named-state queue file miss) are rotated to the bottom of the runnable pool. A thread is promoted from the stalled pool to the runnable

pool when the thread's data arrives via the network interface. All incoming data must go through the network interface because the only mechanism for data to be delivered to a thread is via queue mappings, and all queue mappings are routed through the network interface.

A dedicated scheduler and profiling co-processor is provided in the Q-Machine to remove the overhead of figuring out which threads to run and when to migrate objects from the execution core. The scheduler works only on locally available information and runs out of a small bank of local memory, so its implementation is much lighter-weight than the execution core. In other words, the scheduler does not require the queue-based inter-thread communication mechanisms implemented in the execution core, so it can use a simpler register file and direct load/store memory access mechanisms. Thus, the scheduler is implemented as a slightly enhanced 16- or 32-bit RISC processor that runs entirely out of a few megabits of local memory. In an implementation taped out in 2010 – more on this in section 5.4 – a memory of 5 Megabits is presumed to be very easily implemented in fast SRAM technology; if DRAM technology is used, the capacity could be 10 or 20 times that amount. A programmable scheduler co-processor is chosen over dedicated hardware because the scheduling and migration problem is very complex and difficult to implement directly in hardware. Also, an ambitious user or a compiler may wish to tune the scheduler code if very regular or predictable thread running patterns are expected.

The scheduler co-processor's primary hardware enhancement is a direct interface to the empty-full bits of threads pending scheduling; incoming data from inter-thread traffic must go through the scheduler before being written into the PQF (or to the environment cache if the PQF is full), so that the scheduler knows which threads have become un-blocked as a result of the arrival of pending data. The other hardware enhancement of the scheduler

is a fast direct interface to the scheduler list. The scheduler list is intimately tied the instruction cache. Each scheduler list entry contains an I-cache line, a finger into the line that reflects the exact program counter value, a context ID, and a pointer field that, if valid, contains a pointer to the next scheduler list entry. Scheduler list entries that are unused can be treated as generic I-cache lines and vice-versa. An example of the hybrid scheduler/I-cache structure is illustrated in figure 5.4. In order to keep I-cache speeds high, a smaller cache-buffer may be employed that is dedicated only to I-caching, or perhaps the implementor can separate the tag and context ID fields for both functions and use a less associative but faster comparison for lines that are marked as tagged. The scheduler only performs index-based lookups for scheduler items, so it does not require an associative comparator, but rather requires a longer tag field, since multiple threads will often run through the same piece of code. Note that the I-cache and scheduler functions can be made mutually exclusive while sharing the same physical space without too much of an impact on the critical I-cache indexing and lookup path: the cache comparison lines can have the "match" output of the comparator gated by the "sched" mode bit; thus lines devoted to scheduler functionality look just like invalid lines to the caching function.

It is anticipated that the instruction cache will have a capacity of several thousand lines, so in order for the node to enter scheduler-lock, there must be around several thousand threads to run. If the capacity of the scheduler structure is deemed to be too small to fit in hardware, an extra bit can be provided in the "next runnable" field that causes the scheduler co-processor to take a trap when requesting the next runnable line and to instead check an explicitly managed main memory-based linked list.

The hybrid scheduler/list structure prefers to have at least two write and two read ports, one each for the scheduler function and for the I-cache func-

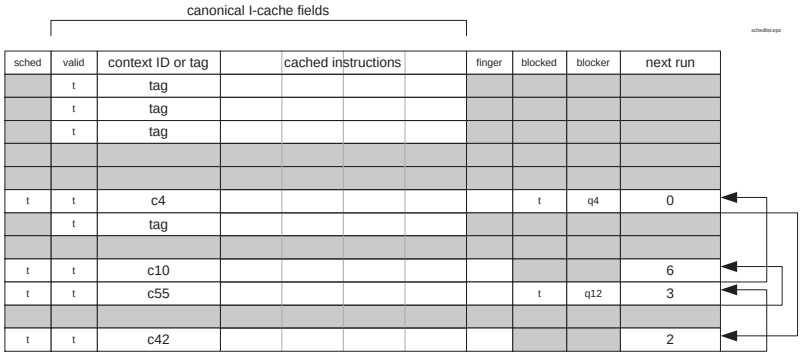


Figure 5.4: Hybrid scheduler list/I-cache structure. In this diagram, c42 and c10 are runnable and up for forwarding to the work-queue; as values for c55:q12 and c4:q4 arrive via the NI, they will be promoted to runnable status.

tion; however, an implementation can get away with single read/write ports if short stalls are tolerable. Although at first glance scheduler traffic may seem to be very small compared to that of cache line traffic, the scheduler co-processor is also responsible for modifying the order of the linked list of runnable items depending on the item's priority and status. It is also responsible for inserting and deleting scheduled items as threads are spawned, garbage-collected, or migrated in and out of the node.

Network Interface

A discussion of the network interface used in the Q-Machine is deferred to appendix C.

5.3.2 Memory Node

The memory node is implemented as a network interface to a large bank of DRAM plus a small co-processor that helps coordinate concurrent and atomic operations. It also can help manage a local data only cache to improve access times and to increase the average number of requests responded per network

cycle. The network interface used by the memory node is identical to that used by the processor node.

The method of accessing memory in the ADAM model is to first send an initial access capability to a memory node, and subsequently send only offsets to that capability. If another capability is seen coming in from an already initialized context ID/queue pair, it is interpreted as a re-initialization of the access capability for that pair. Note that context ID/queue pairs are unique throughout the entire machine by design. As threads are garbage collected or memory mappings explicitly destroyed, access capabilities are removed from the access table.

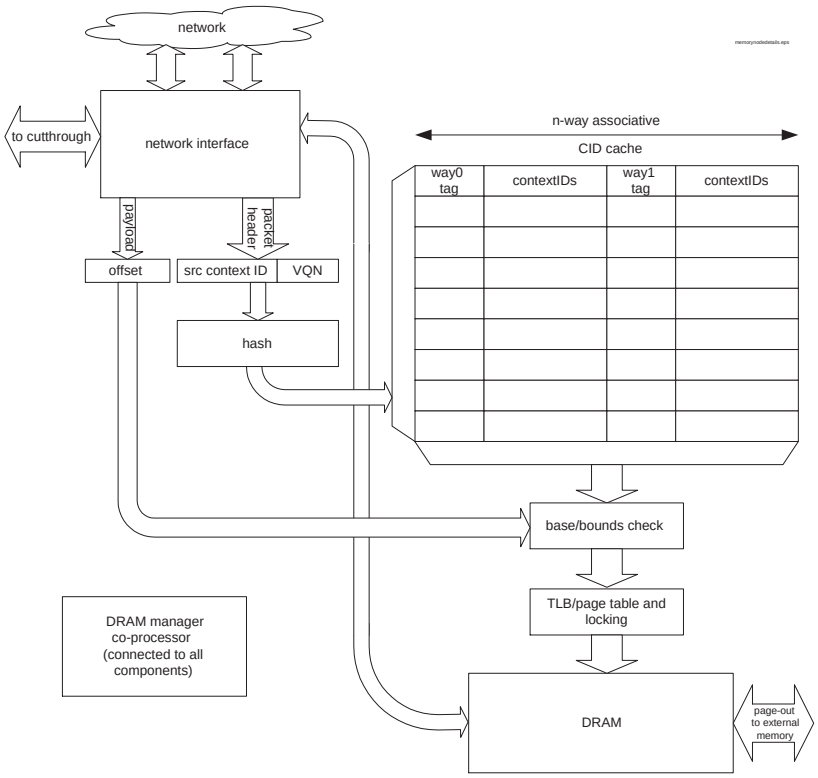


Figure 5.5: High level block diagram of a memory node.

The access table is implemented exactly like a cache; see figure 5.5. The index and tag into the access table is determined by hashing the requesting context ID and queue number pair. The requester identification information is embedded into every network packet (including packets that come over the cut-through interface). The implementor is free to optimize the cache size and associativity, along with the hash function, to optimize the hit rate. In the case of a cache collision, the replaced data cannot be thrown away, as in a cache; instead, the data has to be retired to storage that the memory co-processor manages.

The memory co-processor is also responsible for managing atomic transactions. When a thread executes an `EXCH` instruction, a packet is sent to the memory node that marks the access capability as atomic. The next time the thread attempts to access a queue that maps to the `EXCH` queue, the thread (actually, the processor node that the thread is running on) negotiates a lock on the capability and performs the atomic exchange. It is one of the memory co-processor's duties to coordinate this locking feature.

The address space allocated to each memory node is much larger than the implemented memory at the memory node; thus, it is assumed that every memory node has access to a slower but very large backing storage, be it a disk drive, or conventional DRAM backed by a disk drive. Paging is done in a conventional manner, and the memory co-processor is responsible for managing paging as well.

The ADAM System Simulator implements a memory node as an array with uniform lumped average access latency.

5.4 Physical Design

I describe here what a physical implementation of the Q-Machine might look like. This exercise is an important step in grounding the simulator parameters in some semblance of reality; readers are invited to skip this section if they have little interest in physical design.

5.4.1 Technology Assumptions

Before delving into the details of the Q-Machine physical implementation, I will summarize my key technology assumptions. I assume that the final implementation of the Q-Machine will be a medium to large machine consisting of an array of tile processor chips. The number of nodes represented by the entire array is expected to be in the range of 1,000 for a desktop machine to 1,000,000 for a room-sized supercomputer. I also assume the availability of wall outlet power and perhaps a liquid cooling scheme employing microchannels [Tuc84] to give a maximum thermal budget of at least 1000 watts per chip (actual consumption is assumed to be much less than this). I also presume that the implementation will tape out around 2010, give or take a couple of years. One of the more significant aspects of ADAM is that it can leverage the upcoming higher level of integration while coping with the wire delays, complexity and yield issues commonly anticipated to be problems with 2010-level process technology, while maintaining a backward compatible path with ADAM implementations built today. Table 5.1 summarizes the technology that might be available in 2010.

Note that the [AHKB00] data is based on the Semiconductor Industry Association (SIA) 1999 roadmap which presumes a chip area of about 800 mm^2 at the 50 nm node, whereas all of the other data is pulled from the updated SIA 2000 roadmap. The availability of a 3-D CMOS process is not addressed by

Parameter	Value
Lithography	50 nm
Gate Length	$\sim$30 nm
Layers of metal	10 minimum
Short wire pitch	100 nm [CI00b]
Short wire maximum run	300 μm [CI00b]
Chip size (production)	400 mm^2
Chip size (maximum)	572 mm^2
Logic density, auto-layout ASIC	400-800 Mtransistors/cm^2
SRAM density, high-performance	1423 Mtransistors/cm^2 or 237 Mbits/cm^2
Maximum SRAM cache size @0.3ns access time	$\sim$4 KB [AHKB00]
Maximum SRAM cache size @0.5ns access time	$\sim$100 KB [AHKB00]
Maximum SRAM cache size @1.0ns access time	$\sim$1000 KB [AHKB00]
Anticipated memory to logic ratio	9:1 [CI00c]
DRAM cell size, optimized	0.0064 μm^2, or 15.6 Gbits/cm^2
Clock rate, ASIC (cross-chip)	1.5 GHz
Clock rate, local	10 GHz
Clock rate, 16FO4 delays per clock	3.5 GHz [AHKB00]
Reachable chip area in 1ns (16FO4 delays)	about 10% [AHKB00]
Signal I/O pads available	2700
Chip-board signaling rate	3.1 GHz
ASIC defects D_o, D/m^2 (65% yield)	787
Cost, at introduction, using high-performance (large on-chip memory) CPU model	3.8 μcents/transistor
Number of silicon layers	At least 2
Interconnect pitch between layers	Equivalent to top-level metal

Table 5.1: Extrapolated Technology Parameters for 2010. All values from [CI00a] unless otherwise noted.

the SIA roadmap, but a number of companies and research labs have shown promising results, such as Matrix Semiconductor and MIT Lincoln Labs. Matrix Semiconductor has demonstrated multilevel silicon devices that were fabricated on a TSMC process. Their basic approach is to deposit thin films of amorphous silicon on planarized dielectric layers, and then to anneal the silicon into crystals large enough to form transistors. Their approach does not necessarily yield high-performance logic, but it does provide hopes for a high-density memory. [Sem] MIT Lincoln Labs' approach, on the other hand, can yield at least two layers of high-performance logic. Their approach bonds two SOI CMOS wafers or chips together face-to-face using hydrophilic room temperature bonding. To create more layers of logic, one or both of the bonded wafers can be thinned using an etch and/or Chemical Mechanical Polishing (CMP) process, relying on the SiO_2 layer as an etch-stop. Another layer of logic can be hydrophilically bonded and the process repeated. [LBCF+00]

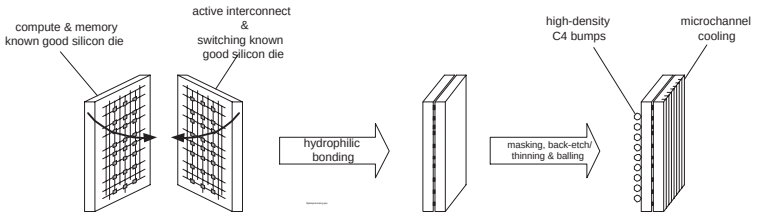


Figure 5.6: Packaging and integration for a two-layer silicon high-performance chip multiprocessor.

This admittedly fuzzy look into the cloudy crystal-ball of the future forms the basis for some of the constants associated with the Q-Machine implementation. It is important to reiterate that the ADAM does not rely on any of this technology coming to pass; one could implement ADAM in today's technology.

5.4.2 Design Description

The Q-Machine physical design uses a high-performance two-layer silicon process, as illustrated in figure 5.6. One layer is dedicated to the processor nodes, and the other layer is dedicated to the active switching network. Each layer can be independently tested before integration via built-in-self-test(BIST) and wafer probing to help boost system yields.

There are several advantages to this partitioning of the design into network and processor layers. By giving an entire layer to the active switching network, the interconnect can use fatter, wider-spaced differential wires and buffer placement is less constrained. In addition, the interconnect layer contains all of the routers and switches. Finally, the interconnect layer is entirely generic: the user is free to re-use the interconnect layer across several designs and incorporate custom nodes on the processor layer. An architecture that leverages this kind of reconfigurability is described by [CCH+00]. The obvious advantage for the processor layer is that it is free of the overhead of network wiring and buffering, and thus it has fewer constraints on the size, layout and placement of the nodes. A schematic of what the network layer may look like is presented in figure 5.7. A discussion of the topology of the network chosen for this implementation can be found in section C.3.

The processor layer for this implementation is chosen to be a simple tile format. Each unit tile consists of a memory node and a processor node (an architectural block diagram of the processor node can be found at figure 5.2 with a description in section 5.3). The memory node and the processor node are laid out as tori around the network interface (NI). This toroidal arrangement around the network interface helps minimize the worst-case distance of any of the slower wires used on the processor layer to the faster interconnect on the network layer. An overview of what the fully-tiled node might look like can be found in figure 5.9.

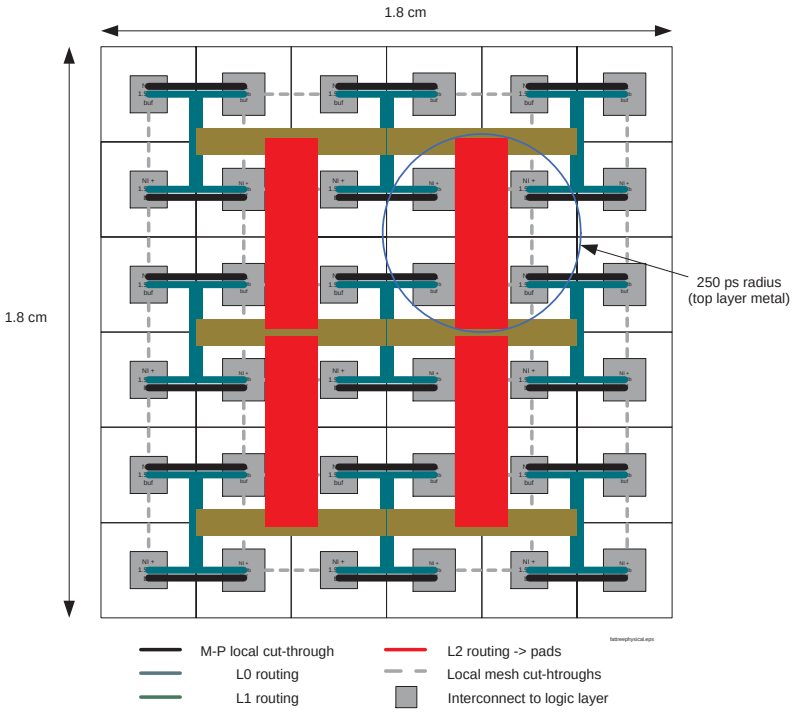


Figure 5.7: Cartoon of the network layer layout.

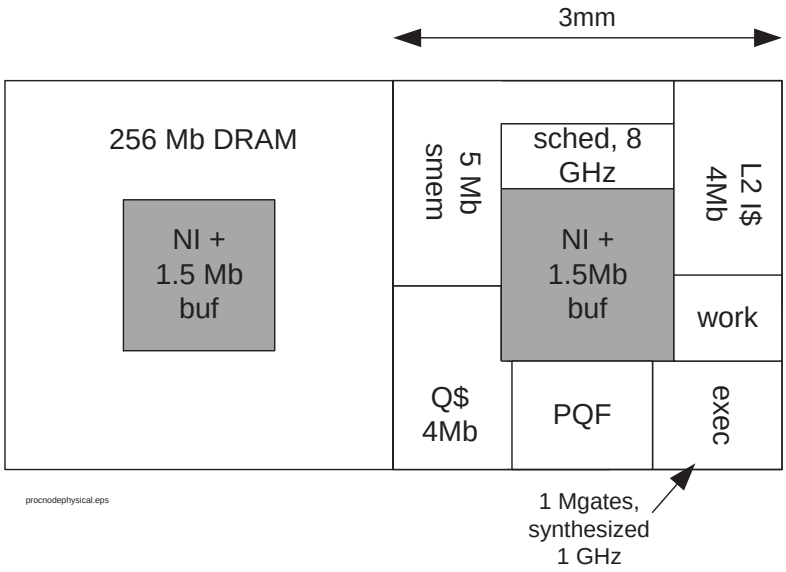


Figure 5.8: Hypothetical layout of a single processor node.

The anticipated clock rate of a single processor node is 1 GHz. This number is derived by looking at the radius of communication over a 1000 ps interval and the estimated clock period for logic built using a design rule of 64 FO4 inverter levels at the 35 nm process node. [AHKB00] The relatively relaxed 64 FO4 inverter levels criteria was chosen in order to allow the processor design to be accomplished with fewer pipeline stages and a primarily synthesized verilog design methodology with a few well-chosen hand-optimized blocks (such as the multipliers, adders and barrel shifters). Simply stated, the assumptions about the physical implementation of this architecture my thesis are kept very conservative, to help compensate for their extremely speculative nature. Also, the actual performance of the migration mechanism in my thesis is always quoted in terms of network cycles, and compared against other implementations by normalizing cited times to clock cycles. Normalizing to

clock cycles helps to factor out technology assumptions. Finally, since the performance of the migration mechanism in this architecture is dominated by the performance of the network, the actual clock rate of the processor could be much higher and have little impact on the results of this thesis. The detailed assumptions about the network interface are given in appendix C. The short summary is that the network interface should be able to send, in the worst case, one flit every 500 ps to 1 ns.

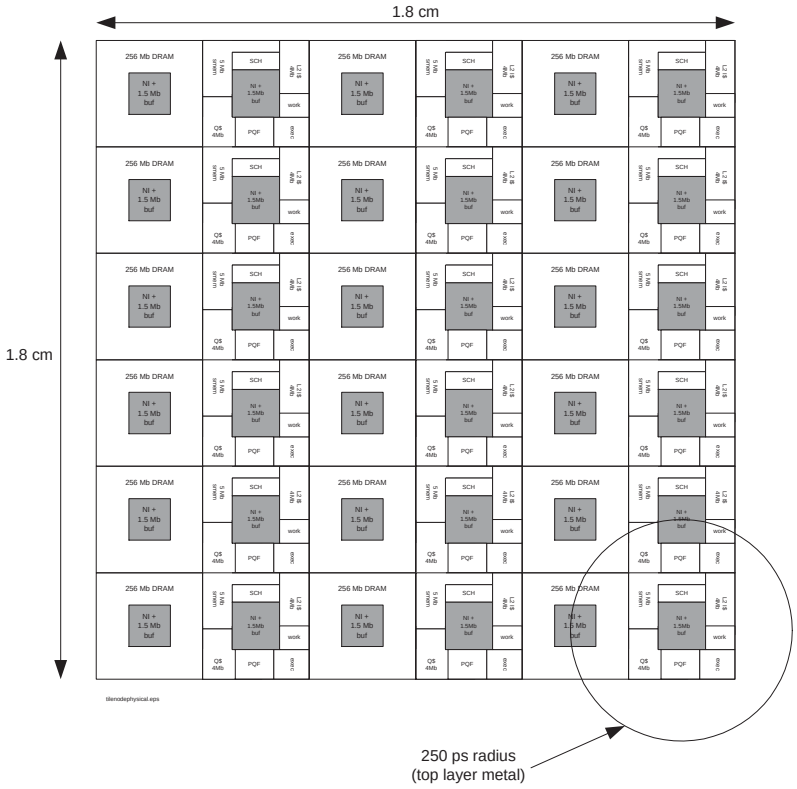


Figure 5.9: Hypothetical layout of the tile processor chip.

Chapter 6

Machine and Migration Characterization

> 42.7% of all statistics are made up on the spot.
>
> —*The Hon. W. Richard Walton, Sr.*

The last section described a fast, low-overhead mechanism for moving data and threads around within the Q-Machine implementation of ADAM. This section summarizes the basic performance characteristics of the Q-Machine and present the results and analysis of several benchmarks.

6.1 Basic Q-Machine Performance Results

This section presents some basic performance characteristics of the Q-Machine and its network as implemented by Adam System Simulator. All instructions are assumed to complete at a rate of one per cycle, as long as its dependencies

are satisfied. An instruction that has been scheduled, but is missing a dependency, causes a single-cycle bubble to be inserted into the execution stream. Future implementations could design the PQF to interact with the scheduler in such a manner that this bubble is eliminated, however this generation simulator was written to maximally simplify processor node implementation. As mentioned previously, future implementations could also enhance the processor node's performance by adding out of order, superscalar issue, or SMT to the core. The latter two features are eschewed because they would would require more queue file ports; the former would require an associative lookup within the pending out of order issue window in a thread whenever a piece of data arrived from the network interface.

The Adam System Simulator (ASS) used to derive all the results for my thesis implements the full idempotent source-responsible network protocol described in section C.2, and simulates the network in a cycle-accurate fashion. The network topology is a radix-4 dilation-2 randomly wired fat tree with $\frac{1}{2}$ bandwidth scaling per tree level; all wiring is unidirectional point-to-point. Each processor or memory node has four ports into the network: two in, and two out. The simulated processor nodes also take into account the swapping mechanisms required for the named-state queue file (as described in appendix C). The simulator also takes into account stalls due to memory bandwidth limitations. The latency of the network interface in all modes of routing (cut-through, loopback, and off-node routing) is also accounted for by the simulator. The simulator was written in Java and achieves a peak simulation rate of around 20 kcycles/s aggregate on a dual Athlon XP 1900+ system. A screenshot of ASS can be seen in figure 6.1.

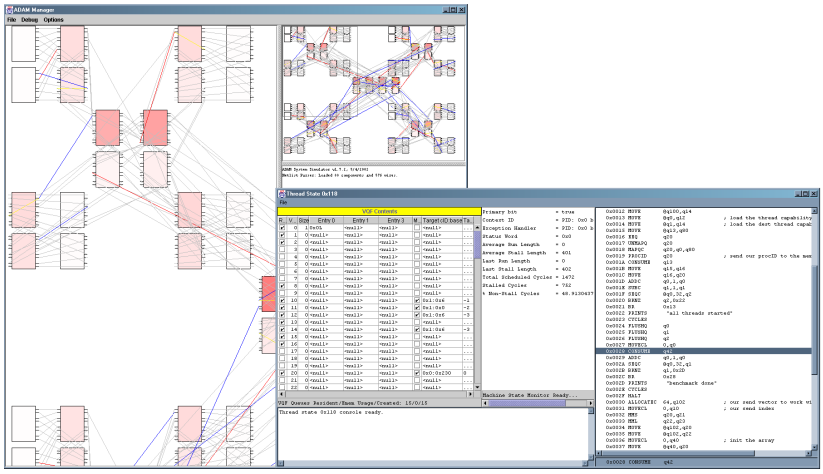


Figure 6.1: Screenshot of the ASS running a 64-node vector reverse regression test. On the left is the machine overview; to the right is the thread debugger window.

6.1.1 Memory Performance

These results summarize the essential processor node to preferred memory node access times. All of the following results are for an unloaded machine running only the test code.

- **Allocation Latency:** 10 cycles from execution of the `ALLOCATE` instruction to issue of its dependent instruction. The allocation algorithm in the simulator is simple; it just increments an allocation pointer and returns a capability of the desired size.

- **Load from local memory:** 7 cycles from the execution of the `MOVE` instruction that sends the load address to the issue of an instruction dependent upon the load result. The breakdown of the load timing is 1 cycle from processor core → NI; 1 cycle from NI cut-through port → preferred memory node; 2 cycles to perform memory access; 1 cycle

memory → memory node NI; 1 cycle from memory node NI → processor; 1 cycle to re-schedule and re-issue the dependent instruction.

- **Store to local memory:** 6 cycles from the execution of the `MOVE` instruction that satisfies the atomic address and data tuple to the issue of the dependent instruction. The latency breakdown of a store is similar to that of a load, except that only 1 cycle is spent in memory because the store acknowledge return can be overlapped with the store.

These numbers are conservative for the target process technology; one cycle is budgeted each way for wire delays due to the anticipated spacing of the memory from the processor.

6.1.2 Basic Network Operations Performance

The general formula for the latency of a thread-to-thread communication packet (L_n) is:

$$L_n = 2 * T_{proc} + 2 * T_{route} * n + (L_{packet} - 1) \tag{6.1}$$

where

T_{proc} is the processor node network interface overhead of 2 cycles

T_{route} is the time required to traverse a router, which is 3 cycles

n is the route depth, equal to the number of levels up the tree a packet must travel

L_{packet} is the length of the data packet, which is 4 for a short data packet The following tests were run on an unloaded 16-processor simulation; these tests confirm the validity of equation 6.1.

- **Loopback latency:** Loopback latency is the time required for a thread to communicate with another thread on the same processor node. This

time is 4 cycles from producer's execution to the issue of the consuming instruction under minimum scheduler loading. The breakdown of the 4 cycles is: 1 cycle from the processor core $\rightarrow$ NI; 1 cycle to identify and process the loopback; 1 cycle from the NI $\rightarrow$ PQF write port; 1 cycle to re-schedule and re-issue the dependent instruction. Latencies under real workloads will typically include some amount of time spent servicing other threads in the work queue.

- **Thread to thread time:** The latency of sending one piece of data to a node that is one up-route away is 14 cycles, not counting issue and execution times of the producer and consumer threads. Hence, there is a roughly 14 cycle one-way latency to the nearest neighbor. The latency breakdown is as follows (obtained through measurements):
  - 2 cycles after producer execution to push packet to NI
  - 7 cycles through the network (first contact to network to first contact at destination latency) = 3 cycles/router + 1 wire cycle between routers
  - 2 cycles for the tail of the packet to "catch up"
  - 3 cycles to collate and issue the consumer instruction

- **Remote Load Access Latency, Full Diameter on a 16-node Machine:** 55 cycles round-trip latency

6.2 Migration Performance and Migration Control: Simple Cases

This section presents the results of applying the migration mechanisms to two simple cases: two threads communicating exclusively with each other, and

a thread and memory communicating exclusively with each other. A brief formal analysis is also performed to determine the optimal omniscient and optimal on-line algorithms for controlling migration in these cases.

6.2.1 Two Threads Benchmark

This benchmark is used to determine the thread migration overhead. A tight loop of dependent operations between two threads is constructed; the difference in the time per message loop during thread migration and during normal operation is the migration overhead. Specifically, two threads communicate exclusively with each other. Each thread is initialized with a unique token; the tokens are swapped between the threads and incremented over 32 iterations. On the fifth iteration, a manual `MIGRATE` instruction is issued which forces one thread to migrate toward its partner. The use of a manually invoked migration allows greater control over the benchmarking process. There is no reduction of processor overhead by manually controlling migration since this task is typically handled by the scheduling and profiling coprocessor. A diagram of the communication pattern can be found in figure 6.2, and the code for this synthetic benchmark can be found in figure 6.3.

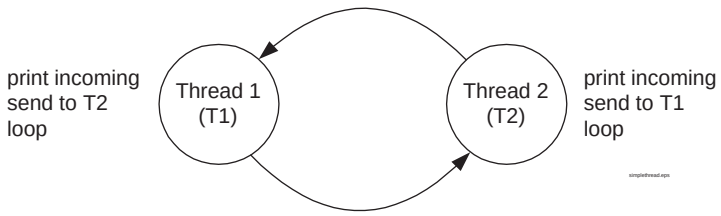


Figure 6.2: The two threads synthetic benchmark. Communication happens along the arcs; a data dependency is forced by printing the incoming data.

Two-Thread Benchmark Results

```
main:
        MOVECC  0, q100       ; spawn one thread local
        MOVECC  4, q101       ; spawn one thread distance 4 away
        SPAWNC  q100, thread1, q0
        SPAWNC  q101, thread2, q1
        MAPQC   q10, q0, @q0
        MAPQC   q11, q0, @q1
        MOVE    @q0, q11      ; thread 1, meet thread 2
        MOVE    @q1, q10      ; thread 2, meet thread 1
        HALT                  ; my work is done
thread1:
        MOVECL 0, q10
        MAPQC  q20, q1, @q0
loop1:
        MOVE   @q10, q20
        PRINTQX q1
        SEQC   @q10, 0x20, q30
        SEQC   @q10, 0x5, q40 ; this segment used to control when
        ADDC   q10, 1, q10    ; migration occurs during testing
        BRZ    q40, byp1
        PROCID q41
        MOVECL 2, q42
        MIGRATE q41, @q0
byp1:
        BRZ    q30, loop1
        CYCLES            ; CYCLES prints out current cycle count
        HALT              ; available only in the sim environment
thread2:
        MOVECL 0x100, q10
        MAPQC  q20, q1, @q0
loop2:
        MOVE   @q10, q20
        PRINTQX q1
        SEQC   @q10, 0x120, q30
        ADDC   q10, 1, q10
        BRZ    q30, loop2
        CYCLES
        HALT
```

Figure 6.3: Code used for the two thread benchmark.

The two threads benchmark with migration was run over five cases that varied the starting position of the threads. These trials were compared to the two threads benchmark run without migration over the same five starting positions. The benchmark yielded the following results:

- The measured time overhead of a lightweight migration over a distance of two up-routes is 66 cycles. Time overheads are computed as the time added to a single iteration result when compared to the non-migrated case.
- The measured time overhead of a heavyweight migration over a distance of two up-routes is 78 cycles; a heavyweight migration was forced by tweaking the internal simulator heavy/light decision threshold.
- Benchmark speedup scales linearly with migration distance (figure 6.4). Speedups are bigger in a real system implementation because the simulation environment assumes that wire delays between tree levels are constant, regardless of the size of the tree. In other words, a real implementation will have more wire delay, especially in a large implementation, and the impact of migration will be greater.
- The zero-distance case in figure 6.4 shows lower than unity performance because there is a slight 14-cycle overhead for executing a migrate command manually, even if it does not move the thread.

The estimated system latency of an L2 cache fill on a Pentium 4 is about 175 ns (which is 140 800 MHz Direct-RAMBUS cycles) [CJDM01]. Thread migration times in my architecture thus compare favorably to an L2 cache fill on a conventional contemporary processor.

Analysis

I will now derive an on-line algorithm for guiding migration decisions in the two-thread scenario, and also determine how many loop iterations must happen in order to amortize the cost of a migration. Formally speaking, the

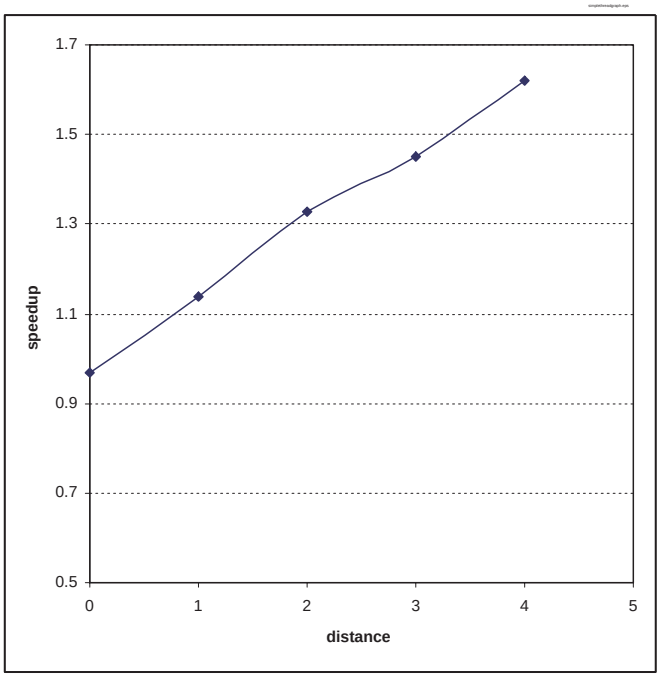


Figure 6.4: Measured speedup versus migration distance for the Two Threads benchmark.

threads communicate using a message sequence, σ. For any given σ, I will derive a migration algorithm, ALGTT, and evaluate its competitiveness with respect to the optimal algorithm, OPTTT.

In the case of the Two Threads microbenchmark, a sequence consists of i messages, m, that each contribute a partial cost as a function of the routing distance d:

$$\sigma_d^i = m_1(d), m_2(d), \ldots m_i(d) \tag{6.2}$$

Let us denote the cost of an algorithm – the time it takes to execute given σ – as ALG(σ). If the cost of moving a thread is $M(d)$, then our algorithms are defined for $d > d'$:

- OPTTT: If $[\text{OPTTT}(\sigma_{d'}) + M(d-d')] - \text{OPTTT}(\sigma_d) < 0$, then migrate thread 1 from d to d' before the first iteration.
- ALGTT: If at iteration i, $\sum_{k=1}^{i}[m_k(d)] = \sum_{k=1}^{i}[m_k(d')] + M(d-d')$, migrate thread 1 from d to d'.

The competitiveness of ALGTT and OPTTT is now derived.

Theorem. *ALGTT is at worst 2-competitive with OPTTT.*

PROOF.

By inspection, OPTTT(σ_d^i) = ALGTT(σ_d^i) for OPTTT$(\sigma_{d'}^i) + M(d-d') \geq$ OPTTT(σ_d^i). Let us define the value of i where OPTTT ceases to be equal to ALGTT as the equivalence point e. In cases where $i > e$, ALGTT's competitive ratio against OPTTT is

$$\frac{\sum_{k=1}^{e}[m_k(d)] + M(d-d') + \sum_{k=e+1}^{i}[m_k(d')]}{\sum_{k=1}^{i}[m_k(d')] + M(d-d')} \tag{6.3}$$

Clearly, the worst case would be if $e = i$, because ALGTT would pay for $M(d-d')$ and never amortize its cost. At this point, the competitive ratio is just

$$\frac{k + M(d - d')}{k} \tag{6.4}$$

where $k = \sum_{k=1}^{e}[m_k(d')] + M(d - d') \equiv \sum_{k=1}^{e}[m_k(d)]$.

Thus, as $\lim[m_i(d')] \rightarrow 0$, ALGTT $\rightarrow 2^{-} \cdot$ OPTTT. ■

Theorem. *ALGTT is an optimal on-line algorithm for the Two Thread microbenchmark.*

PROOF.

There are two cases to consider for ALGTT when comparing against another algorithm, ALG:

MIGRATE EARLIER. In the case that ALG were to migrate earlier than ALGTT, the worst case performance for ALG would be a sequence that ended right at ALG's decision threshold. The competitive ratio in this case would be

$$\frac{\sum_{k=1}^{i}[m_k(d)] + M(d - d')}{\sum_{k=1}^{i}[m_k(d)]} \tag{6.5}$$

One can see that this function is monotonically increasing for decreasing values of $\sum_{k=1}^{i}[m_k(d)]$ (figure 6.5); hence, it is not possible for ALG to have a lower competitive ratio than ALGTT.

MIGRATE LATER. In the case that ALG were to migrate later than ALGTT, the worst case performance for ALG would again be a sequence that ended right at ALG's decision threshold. In this case, the competitive ratio would be

$$\frac{k + \sum_{k=e+1}^{i}[m_k(d)] + M(d - d')}{k + \sum_{k=e+1}^{i}[m_k(d')] + M(d - d')} \tag{6.6}$$

where $k = \sum_{k=1}^{e}[m_k(d)]$, and e is the equivalence point as previously defined. As the decision point of ALG increases beyond e, the numerator of equation 6.3 grows slower than the numerator of equation 6.6 since $m_k(d) >$

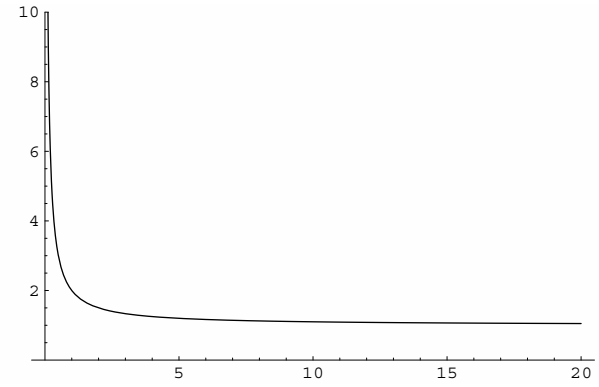


Figure 6.5: Shape of the curve $\frac{x+c}{x}$.

$m_k(d')$. Because the denominators are the same, there is no way that ALG can be less than ALGTT. ■

I will now determine the curve for the equivalence point, e, versus various migration overheads. The point, e, represents where the cost of migrating a thread is amortized by the savings in thread communication time. In order to determine this, I will derive an expression for the message delivery time, $m_i(d)$. The general formula for routing delay in the Q-Machine implementation is

$$m_i(d) = L_R \cdot d \tag{6.7}$$

where L_R is the latency contribution of routers for one tree level and d is the routing distance, *i.e.*, the number of tree levels spanned by the route. In the Q-Machine, $L_R = 6$.

Recall that the equivalence point, e, is defined as

$$e \equiv \sum_{i=1}^{e} m_i(d') + M(d - d') = \sum_{i=1}^{e} m_i(d) \tag{6.8}$$

Rewriting yields

$$e \equiv \sum_{i=1}^{e} m_i(d) - \sum_{i=1}^{e} m_i(d') = M(d - d') \tag{6.9}$$

Substituting in 6.7,

$$e = \frac{M(d - d')}{L_R \cdot (d - d')} \tag{6.10}$$

Given equation 6.10, we can create a set of curves indicating how many iterations are required to amortize the cost of a migration for various migration costs (figure 6.6). The cost of migrating a thread, $M(d)$, is assumed to be constant for d in each of these curves, which is a reasonable assumption because a heavyweight thread migration mechanism is assumed for these graphs. Upon inspection of the curves, it is apparent that the cost of a migration is quickly amortized, even for networks of modest size with *constant* wire delay between tree levels. Migration looks even more attractive in a realistic scenario where wire delays grow at best as the square root or cube root of the number of nodes in the machine.

6.2.2 Thread and Memory Benchmark

The thread and memory benchmark is used to determine the data migration overhead, in a manner similar to the two threads benchmark. In the thread and memory benchmark, a single thread communicates exclusively with a single 16-word or 128-word piece of memory through an exchange mapping. A single location in memory is incremented 32 times by the remote thread using the exchange mechanism; on the fifth iteration, the data is forced to migrate

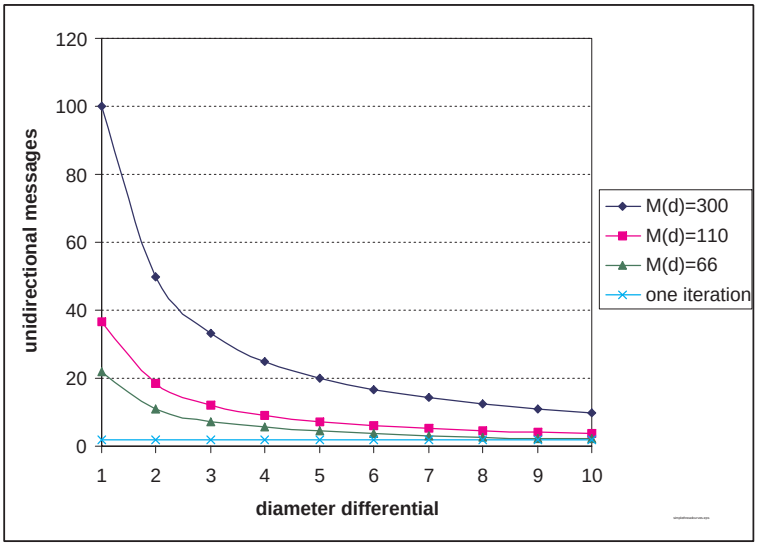


Figure 6.6: Length of message sequence required to amortize various migration overheads ($M(d)$). The baseline two messages per iteration for the Two Thread benchmark is also marked on the graph.

toward the thread. This scenario is similar to the Two Threads case, except that an extra level of indirection is introduced for non-local memory due to the data locater pointer mechanism. The code for this synthetic benchmark can be found in figure 6.8, and a diagram of the communication pattern can be found in figure 6.7.

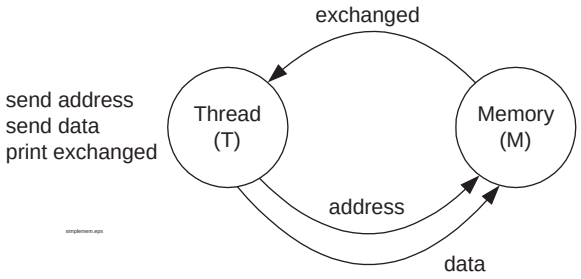


Figure 6.7: The thread and memory synthetic benchmark. Communication happens along the arcs; a data dependency is forced by printing the incoming data.

Thread and Memory Benchmark Results

The Thread-Memory benchmark with migration was run over five cases that varied the starting position of the memory. These trials were compared to the thread-memory benchmark run without migration over the same five starting positions. The benchmark results are summarized in figure 6.9 and figure 6.10.

Figure 6.10 shows the amount of time per iteration versus iteration count, for migration forced on the fifth cycle for the 16-word and 128-word cases. Key data points are labeled with the migration status at that iteration. The amount of migration overhead that is actually experienced by the system is dependent upon the relative timing of the migration request and the incoming memory requests. In the 16-word case, the timing is such that there is virtually no overhead due to memory request freezing and contention during

```
main:
        MOVECC  2, q100         ; spawn one thread distance 2 away
        MOVECC  0, q101         ; allocate memory
        SPAWNC  q100, thread1, q0
        ALLOCATEC  q101, 16, q1
        MAPQC   q10, q0, @q0
        MOVE    @q1, q10        ; thread, meet your memory
        PROCID  q5
        MOVE    q5, q10         ; thread, meet me
        MIGRATE @q1, q20
        ; CONSUME q20           ; substitute for migrate
        PRINTS "migrating"      ; to disable mig.
        HALT                    ; my work is done
thread1:
        EXCH   q20, q21, q22    ; declare exchange queues
        MOVECL 0, q10
        MOVE   q0, q1           ; store our capability in q1
        MOVE   @q1, q20         ; initialize the exchange tuple
        MAPQC  q6, q20, q0      ; map back to our caller...
loop1:
        MOVECL 0, q20           ; always use addr 0 for this test
        MOVE   @q10, q21
        PRINTQX q22
        SEQC   @q10, 0x20, q30
        SEQC   @q10, 0x5, q40   ; this is used to control when
        ADDC   q10, 1, q10      ; migration occurs during testing
        BRZ    q40, byp1
        PROCID q5
        MOVE   q5, q6           ; send a packet to our caller...
byp1:
        BRZ    q30, loop1
        CYCLES            ; CYCLES prints out current cycle count
        HALT              ; available only in the sim environment
```

Figure 6.8: Code used for the thread-memory benchmark.

the migration process; since this is the only step that leads to a slowdown relative to the non-migratory case, the migration of small memory objects is almost free. However, migration overhead scales linearly with the size of the data that is being moved, and eventually the process of freezing and moving the capability adds a non-negligible overhead, as can be seen in the case of moving a 128-word capability.

The overheads and message request characteristics for the Thread-Memory case are similar to the Two Thread case; memory migration can be thought of as thread migration, but faster. Hence, the algorithms and analysis from the previous section apply here.

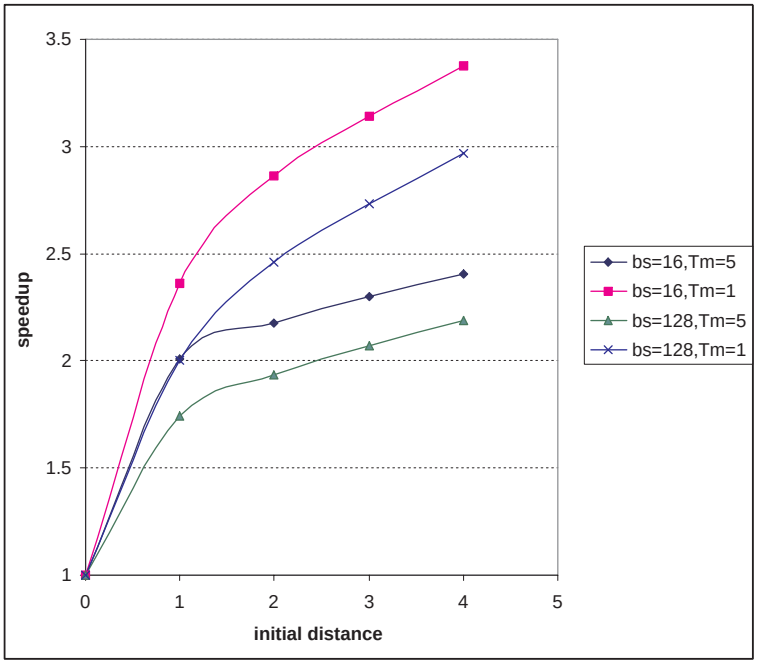


Figure 6.9: Migration speedup versus migration decision time and memory capability size in the thread and memory benchmark.

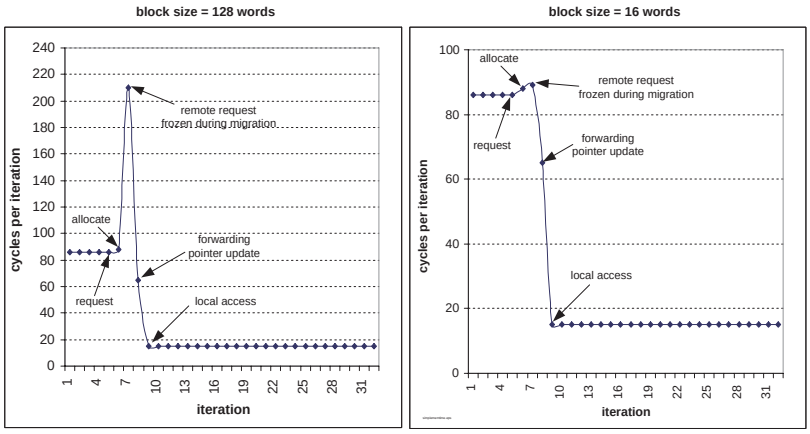


Figure 6.10: Cycles per iteration for Thread-Memory benchmark. $d = 4$ in both cases.

6.3 Application Cases

I will now demonstrate the Q-Machine running some application kernels coded in the People language. These applications are in-place Quicksort, streaming Matrix Multiply, and a simple N-Body gravity simulator. The in-place Quicksort application is used to demonstrate the load balancing abilities of the implementation. The streaming Matrix multiply is used to demonstrate latency-driven data migration, and the N-Body gravity simulation is used to demonstrate latency-driven thread migration. In order to demonstrate my architecture on these kernels, some simple load-balancing and migration algorithms were implemented. For each application, I will briefly provide some background on the load balancing or migration techniques employed, and then present the results of the application benchmark with and without the benefit of dynamic migration control.

6.3.1 In-Place Quicksort Application

A simple in-place Quicksort was written in the People language for this benchmark. Ben Vandiver, the creator of People, wrote the benchmark code. The Quicksort implementation looks very similar to a typical recursive implementation written in C or Java. Figure 6.11 gives a flavor for the Quicksort kernel. Note that in a language like C or Java, this recursive implementation would have little parallelism, as each recursive call to `qsort()` is called in sequence; however, in People, each recursive call actually spawns a new thread. This thread-spawning calling convention introduces latent parallelism in the code that can be uncovered by a load-balancing mechanism.

The load-balancing scheme implemented for this benchmark uses two mechanisms: work-stealing and thread-pushing. Work stealing has been seen before in systems such as Cilk [BL94] and [RSAU91]; thread-pushing is the dynamic work scheduling problem. An overview of dynamic work scheduling algorithms and techniques can be found in [SHK95] and in [XL97].

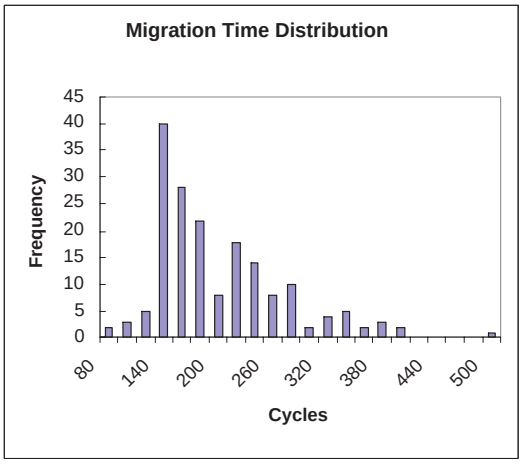
The metric used to determine a processor's load for load balancing purposes is the time, T_w, that the currently running thread spent waiting in the runnable pool. T_w is a direct measure of wasted time because the scheduler only promotes threads to the runnable pool that have its data dependencies resolved. If $\overline{T_m}$ is the expected time required to migrate a thread, work stealing is beneficial if $T_w > T_m$.

In order to implement work stealing, I had to determine $\overline{T_m}$ and I had to implement a load discovery mechanism. $\overline{T_m}$ was determined to be 200 by profiling; the detailed results of the migration profiling can be seen in figure 6.12. A noteworthy observation is that migration times take on a bimodal distribution in the presence of a heavily loaded network. This bimodal distribution is a result of packet collisions in the network. In other words, $\overline{T_m} = 200$ is the expected time given that the migration data packet succeeds

```
int qsort(array[int] arr, int low, int high) {
  if (high == low) {
    return high-low;
  } else {
    int pivot, index, temp;
    int i,j;
    boolean notdone;

    // choose pivot
    index = low +  rand(high-low);
    pivot = arr[index];
    arr[index] = arr[low];
    arr[low] = pivot;

    // partition
    i = low - 1;
    j = high + 1;
    notdone = true;

    while (notdone) {
      while (notdone) {
        j = j - 1;
        notdone = arr[j] > pivot;
      }
      notdone = true;
      while (notdone) {
        i = i + 1;
        notdone = arr[i] < pivot;
      }
      if (i < j) {
        temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;
        notdone = true;
      } else {
        notdone = false;
        index = j;
      }
    }

    membar();

    // recurse
    i = qsort(arr,low,index);
    j = qsort(arr,index+1,high);
    return i + j;
  }
}
```

Figure 6.11: Object method for the Quicksort benchmark written in People.

on the first try. In the case that a migration packet does not succeed on its first try, one could cut the overhead losses and immediately abandon migrating that thread. This would require adjustments to the migration protocol to prevent two copies of the thread from running, though, in the case that the acknowledgment of the migration packet failed to be delivered. This fail-fast migration scheme was not implemented for these simulations, but left as an exercise for future work.



Mean	192
Median	170
Mode	125
Standard Deviation	72
Minimum	58
Maximum	497
Count	177

Figure 6.12: Distribution of migration times used in the Quicksort benchmark

I implemented the load discovery mechanism using periodic discovery queries. The period between queries increases exponentially with the distance between nodes, since the number of queries required grows exponentially with network radius. The base period for discovery queries between neighboring nodes is set to 250 cycles. This period was chosen to be slightly larger than $\overline{T_m}$ in an attempt to provide a sampling period balanced against the expected rate of change in T_w as a result of migration. The response to a discovery query is the processor's time-averaged T_w over the past twenty thread-running events.

Given $\overline{T_m} = 200$, I set the nominal steal threshold at $T_w = 200$ for nearest neighbors. The steal threshold increases linearly with node distance, in order to compensate for the extra routing overhead of reaching farther nodes. The optimal rate of steal threshold increase with distance is probably not linear, but will not affect this benchmark since there is only enough work for two nodes.

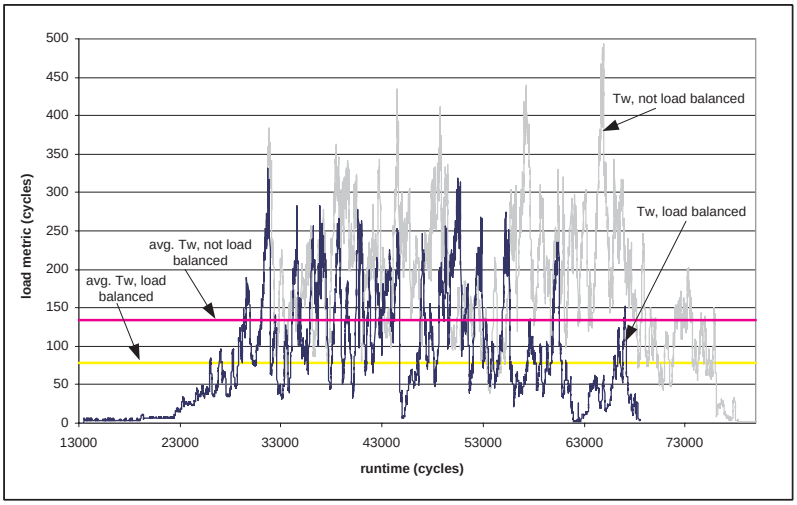


Figure 6.13: Plot of the load metric T_w versus time for the Quicksort benchmark with and without load balancing

The results of the load balancing mechanism on the Quicksort benchmark can be seen in figure 6.13. This figure shows T_w versus time for a Quicksort of 200 elements with and without load balancing using nominal steal metrics. A speedup of 12.3% was observed using the nominal steal threshold; lowering the steal threshold slightly and decreasing the work stealing interval brings the speedup to over 15%. I believe that these more aggressive steal thresholds are not generally a good idea, however; more frequent work discovery packets congests the network and would have a negative impact on performance in

applications with more internode communication.

Thread pushing was also implemented to investigate potential benefits of this mechanism. Thread pushing only happens when a new thread is being created. This kind of thread pushing is trivial to implement on the Q-Machine; it is simply a `SPAWN` instruction targeted at a neighboring node. The danger of thread pushing is that the decision to push is based on stale information; a cluster of nodes can overload a single nearby unloaded node if all the loaded nodes decided to push work onto the unloaded node simultaneously. Even though the danger of overload is small because there is only one source node for threads in this Quicksort benchmark, thread pushing was implemented conservatively. A push only occurs when a neighbor's load metric is observed to be near zero, and the local load metric is observed to be very high, above 400 cycles. In the end, thread pushing was used rarely and accounted for a 1 to 2% speedup in the Quicksort benchmark.

Figure 6.14 illustrates in greater detail the relationship between migration events and T_w. One can see from this figure how T_w is reduced with every migration event. Also shown in this figure is the load incurred on the migration target. Note that this load is kept fairly low throughout the benchmark run.

The Quicksort benchmark demonstrates that the Q-Machine migration implementation is efficient enough to speed up even simple code written with little thought for parallelism. In addition, the migration mechanism is fast enough to provide a speedup on a benchmark that runs for just a few tens of thousands of cycles. In contrast, most other migration mechanisms would take at best a few thousand cycles to complete a single null-thread migration.

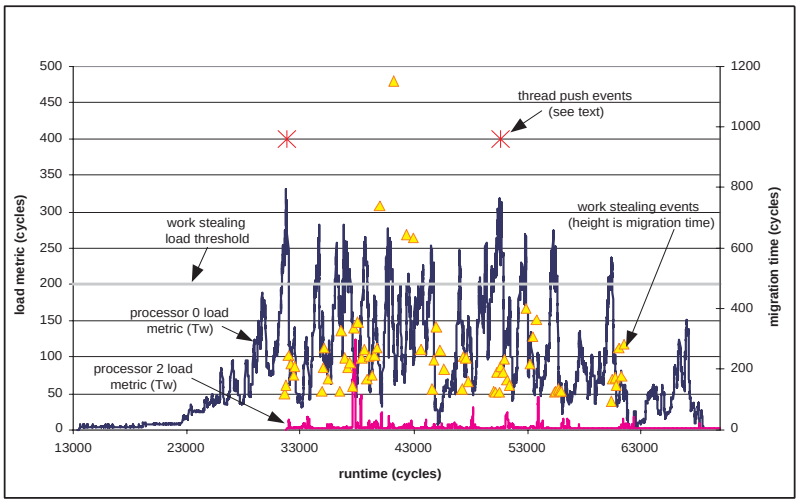


Figure 6.14: Plot of the load balanced Quicksort benchmark with migration events overlayed.

6.3.2 Matrix Multiplication Benchmark

A pair of matrix multiply kernels were written by Ben Vandiver in the People language for this benchmark. The first kernel uses a single nested iterative loop to access the matrix elements and multiply them. The second kernel uses streams to multiply the matrices. Streams are a unique feature of the People language; they are essentially a way of explicitly revealing the underlying queue structures of the architecture to the programmer. The streaming matrix multiply kernel builds two streams that source the matrix multiply data, and a streaming operator that computes and stores the multiply result. These streams allow index computation and array access to happen in parallel with the actual multiply operation. Part of the streaming matrix multiply code is shown in figure 6.15. A stream is called a `module` in People, and its inputs are `sources` and its outputs are `sinks`. The operations `nq()` and `dq()` are

used to enqueue and dequeue data on a stream, respectively.

The standard matrix multiply kernel is used as the reference point in this benchmark; it is a purely single-threaded piece of code. The streaming matrix multiply kernel, on the other hand, instantiates three threads, one each for the matrix sources and one for the multiply operation. Hence, there is an opportunity for data migration to reduce access latencies.

I used a very simple data migration control algorithm in this benchmark. Every 200 cycles, the most popular data element is migrated to the node of the most frequent accesser. The most popular data element is determined by keeping a sorted, rolling list of all accesses over a window of 4000 cycles.

The result of applying this simple migration algorithm is shown in figure 6.16 for a 100x100 matrix multiply, and in figure 6.17 for a 15x15 matrix multiply. One can see that for the 100x100 matrix multiply, the time per iteration drops after the first iteration from around 7,000 cycles to around 1,650 cycles per iteration–about a factor of 4.2 speedup. Note that in figure 6.16, the first iteration time includes the migration overhead for moving two 10,000 element matrices.

On a 15x15 matrix multiply, the migration occurs later, and the time per iteration goes from 1,100 cycles to around 350 cycles. The migration occurs later because the most popular accessors–in this case the matrix multiply stream sources–have to build up "popularity" over the thread that initialized the matrix through a 4000 cycle profiling window. This translates to a speedup of 3.2.

It is also interesting to note that the streaming implementations outperform the single-threaded matrix multiply implementation by about a factor of two in each benchmark case. This is a positive indicator of the performance benefits of the streaming features in the People language. In this specific case, the speedup is a result of parallelizing (decoupling, for those fond of

```
module leftMat has sink[int], source[array[int]], source[int]
  as "vals for left side", "array to use", "size"
  internally source[int] vals, sink[array[int]] arr, sink[int] s {

  int i,j,k;
  array[int] mat = dq(arr);
  int size = dq(s);
  i=0;
  while (i < size) {
    int offset = i*size;
    j=0;
    while (j < size) {
      k=1;
      nq(vals,mat[offset]);
      while (k < size) {
        nq(vals,mat[k+offset]);
        k = k + 1;
      }
      j = j + 1;
    }
    i = i + 1;
  }
}

void matmult(int size, array[int] mat1, array[int] mat2, array[int] mat3) {
  sink[int] lhs,rhs;
  source[array[int]] arr1,arr2;
  source[int] s1,s2;

  construct leftMat with lhs, arr1, s1;
  construct rightMat with rhs, arr2, s2;
  nq(arr1,mat1);
  nq(s1,size);
  nq(arr2,mat2);
  nq(s2,size);

  int i,j,k;
  i=0;
  while (i < size) {
    j=0;
    while (j < size) {
      k=0;
      int sum = 0;
      while (k < size) {
        sum = sum + dq(lhs)*dq(rhs);
        k = k + 1;
      }
      mat3[j+i*size] = sum;
      print(sum);
      j = j + 1;
    }
    i = i + 1;
  }
}
```

Figure 6.15: Portion of the streaming matrix multiply benchmark written in People.

DAE architectures) the memory accesses and the multiply operation.

The per-iteration speedups in the streaming benchmarks are due entirely to the reduction in latency brought about by data migration; load balancing has no impact on the results as the benchmark uses only three threads. One can see in the benchmark results that the multi-threaded streaming implementations without data migration actually perform worse than the single-threaded implementation. This is because People does not account for memory placement with respect to streaming threads, therefore, good performance relies on the availability of a specialized migration mechanism that reduces latency. In this specific instance, the benchmark was run on a 16-node machine, and the source data for each of the streams was migrated across a distance of two router hops each. This move reduces the best-case access latency from 55 cycles down to 7 cycles. Note that another approach to fixing this access latency problem is to use better latency hiding techniques in the code, and to not have the source threads wait for each load to come back from the memory node before forwarding the data onto the streaming multiplier. However, this is a compiler issue, and one of the major points of this benchmark is to demonstrate that data migration can be successfully applied to a user program.

6.3.3 N-Body Benchmark

For this benchmark, I wrote an N-Body gravitational simulator in People. The algorithm used is the basic particle-particle method, where every body computes the net contribution of every other body's force each time step. The second order Runge-Kutta method was used to solve the differential force equation at the heart of the particle-particle method. The numerical core of this code comes from [Che], [Sch] and [Har00]. A graphical representation of the output of the N-Body gravitational simulation can be seen in figure 6.18.

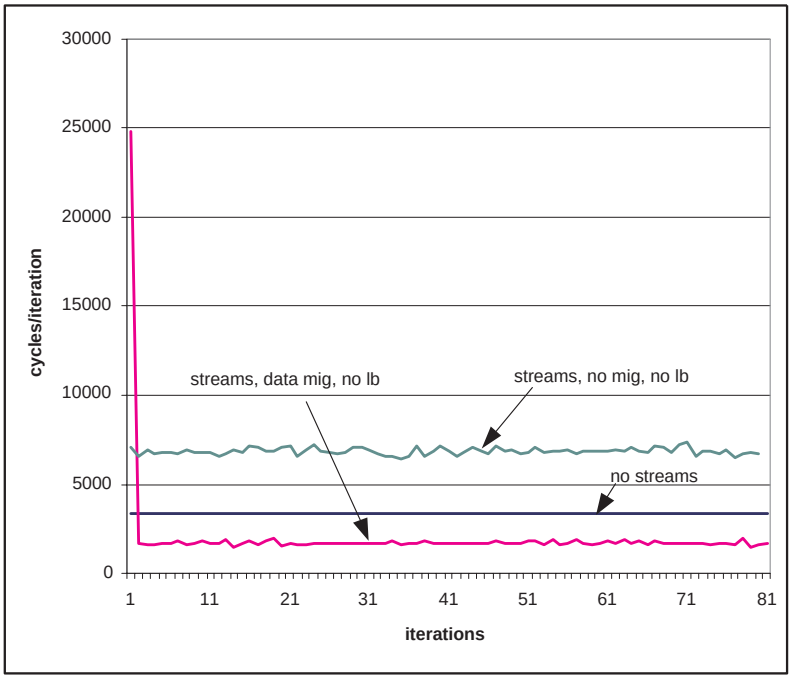


Figure 6.16: Plot of the time required per iteration of a 100x100 matrix multiply over various migration conditions and coding styles.

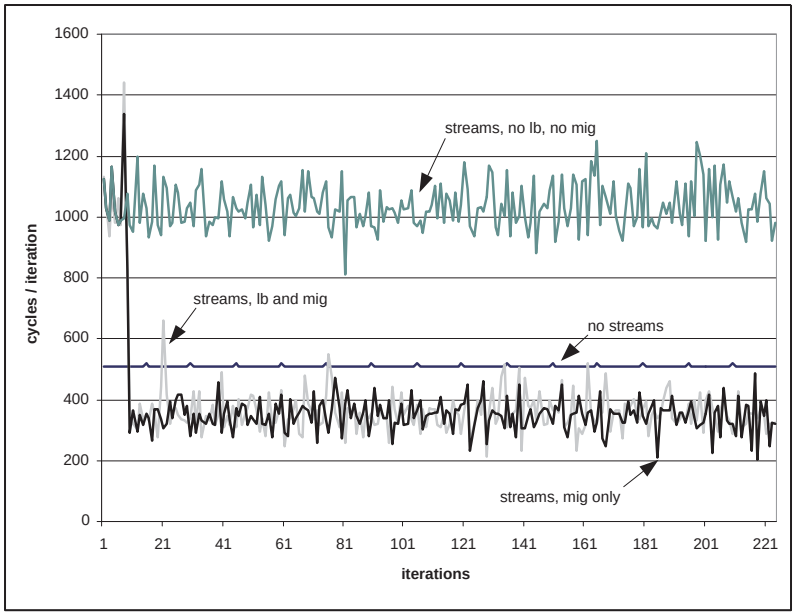


Figure 6.17: Plot of the time required per iteration of a 15x15 matrix multiply over various migration conditions and coding styles.

This figure shows the first few timesteps of a 12-body simulation being run on a 64-node Q-Machine.

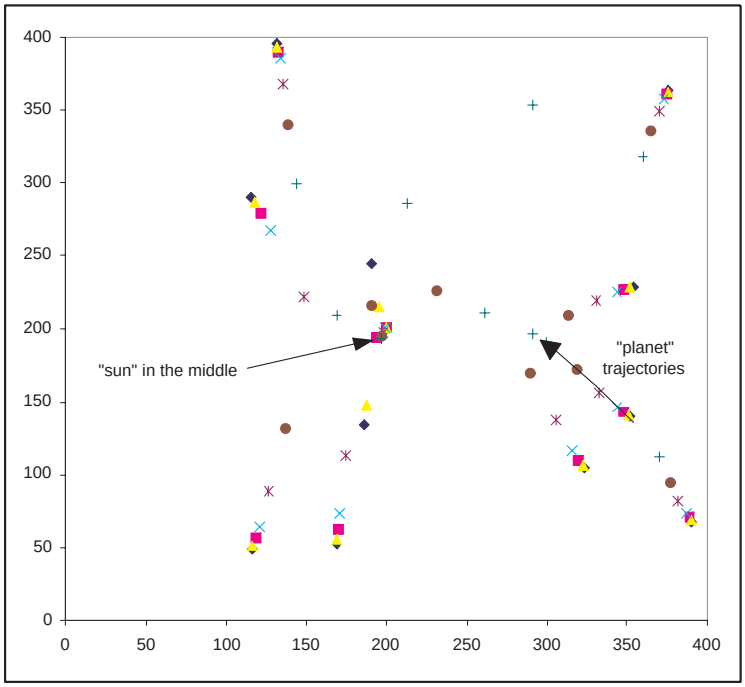


Figure 6.18: Plot of the first few time steps of the N-Body benchmark output

My initial N-Body implementation created a thread per planet and used a binary tree of object-based semaphores to determine the completion of each iteration; Ben Vandiver optimized this to use a tree composed of streams to signal the completion of each iteration. Ben's streaming optimizations plus a few other tweaks reduced the per-iteration time of the N-Body benchmark by about a factor of 4 on its own. Ben's optimizations included static data placement and re-use optimizations, so there is little to gain through dynamic data migration; the only thing that matters is that the initial data be distributed roughly evenly across all the nodes of the machine. Careful initial data place-

ment is important because the actual gravitational force computation is done by an object method invoked by the combining tree, and object methods invocations always spawn near the object's instance variables.

Despite the optimizations, a speedup of 36% was achieved by applying program-guided latency-driven migration to the N-Body benchmark. This result can be seen in the cycles per iteration times plotted in figure 6.20. In order to understand how latency-driven thread migration was used to speed up the N-Body benchmark, one must first understand the structure of the benchmark code.

The inner-loop of the N-Body benchmark consists of a loop that visits each of the planets and computes their partial force contribution on the local planet. Please see figure 6.19. Before entering the inner-loop, all of the instance variables of the local planet object are pulled into temporaries that the compiler holds in queues. These instance variables are written back to the planet object upon exiting the inner-loop. At the beginning of each loop iteration, a remote planet is chosen by the statement `Body body = planets[jj];`. The inner-loop then computes the remote planet's force contribution; during this computation, the loop repeatedly references the remote planet's instance variables. Since the remote planet is typically located on another node, these memory references are fairly slow without any mechanism for reducing access latency. Hence, on the line immediately following the `body` initialization, I call the `migrate(Body)` system function. This function causes the local thread to immediately deschedule itself and migrate itself to the home node of the argument. This `migrate()` call decreases the inner-loop computation time by 36%. This speedup is due entirely to the reduction in access latency to the remote planet's instance variables. Hence, the speedup is proportional to the number of remote instance variable references within the inner loop. For example, simplifying the N-Body differential equation solver to use a

```
float ax=this.ax;  // load up the local planet vars in queues
float ay=this.ay;
// etc...

int jj = 0;
while( jj < size ) {
  if((myIndex != jj)) {
    Body body = planets[jj];  // access a remote planet
    migrate(body);       // migrate ``this'' to planet's node

    rad = (x-body.x)*(x-body.x) + (y-body.y)*(y-body.y);

    // Runge-Kutta kernel omitted for clarity...

    ax = (ax1 + ax2) / 2.0;
    ay = (ay1 + ay2) / 2.0;
  }
  jj = jj + 1;
}

this.ax = ax;   // write back the local planet variables
this.ay = ay;
// etc...
```

Figure 6.19: Inner-loop of N-Body benchmark code.

less accurate but faster Euler method causes thread migration to be ineffective because only three or four instance variable accesses are required in the Euler method. In contrast, the second-order Runge-Kutta method used to derive these results require nineteen instance variable accesses. Note that data migration is never an effective method for speeding up the N-Body application as written because at any given time, multiple threads located on multiple nodes are accessing an object's instance variables.

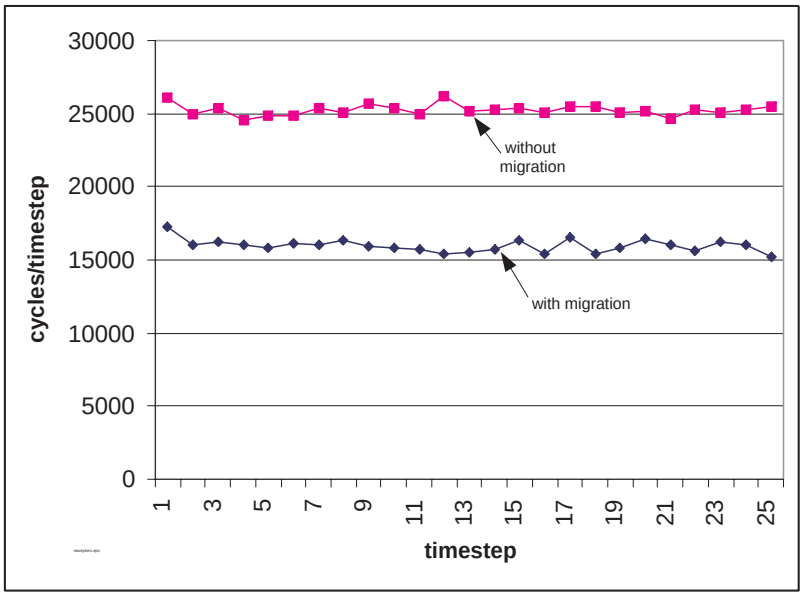


Figure 6.20: Plot of the time required per timestep of a 12-body N-body simulation run on a 64-node Q-Machine.

Chapter 7

Conclusions and Future Work

> ...but I *like* big wrenches. Who cares if there's no bolt big enough for this wrench? Someone will make one someday.
>
> —*Dominic Rizzo*

7.1 Conclusions

This thesis described and demonstrated an abstract machine architecture, the ADAM, that enables high-performance migration mechanisms in hardware. ADAM's architecture features ubiquitous queues that serve as a flexible, uniform hardware abstraction for thread and memory communication. These queues also serve to decouple thread and memory timings. The ADAM architecture also features a capability-based memory system, which enables the fast resolution of data bounds and the enforcement of bounds checks in

hardware. Finally, the architecture features a massively multithreaded programming model where threads are simply special cases of data capabilities, so that a mechanism similar to that used to migrate data can be applied to threads as well. The massively multithreaded nature of the architecture also serves to hide latency by context switching on thread stall events.

The features of the ADAM architecture conspire to enable an efficient and fast migration mechanism for data and threads. This migration mechanism relies on two protocols implemented in hardware: remote data lookup via data locater pointers, and temporally bidirectional pointers for pointer updates after migration events. The migration mechanism itself is fairly simple; in essence, the algorithm is "freeze, copy, and forward".

My migration mechanism's performance was demonstrated in several benchmarks. These benchmarks were run on a predominantly cycle-accurate machine simulator written in Java. The benchmarks were chosen to feature both the load-balancing and the latency-reduction abilities of the implementation. An in-place Quicksort benchmark showed a 12.3% performance increase with simple work-stealing and thread-pushing load balancing algorithms. A streaming matrix multiply benchmark showed a performance increase of 4 times when latency-driven data migration was applied. Finally, an N-Body gravity simulation benchmark demonstrated a speedup of 36% when latency-driven thread migration was applied.

My migration mechanism performs orders of magnitude faster than previous work. Active Messages [WGQH98], a high performance software implementation of thread migration, can push-migrate 6000 null threads per second; this translates to about 300 μs per migration event, or about 16,000 processor cycles at the paper's 50 MHz processor clock speed. My architecture can push-migrate null threads in around 70 cycles, which translates to about 70 ns per migration event in the proposed hardware implementa-

tion. This translates to about a factor of 5000 speedup. My migration mechanism performance is fast enough that it is attractive even when compared against traditional latency hiding mechanisms such as caches: a Pentium 4 L2 cache fill takes about 175 ns, or 140 RAMBUS clock cycles [CJDM01]. The performance of my thesis' migration scheme hopefully makes it an option for latency management in future high-performance parallel computer implementations.

7.2 Future Work

There are many interesting avenues to explore for future work. The most important issue to address will be algorithms for effectively controlling the migration mechanism. Other issues requiring attention will be programming languages and development environments for the architecture. Finally, the architecture needs to be reduced to practice with a real hardware implementation.

7.2.1 Improved Migration Control Algorithms

My work describes some simple metrics and algorithms for controlling the migration mechanism. While these algorithms performed well enough to show a healthy performance increase on several benchmarks, they are far from optimal or complete. My experience with my simple migration control algorithms indicate that these issues will be important during the implementation and testing of future control algorithms:

- **Metrics.** Metrics are required to summarize communication latency and processor load to a control algorithm. Understanding these metrics and choosing the right data to report is a prerequisite to making intelligent control decisions. A good metric should also be implementation-independent

and scalable with technology.

- **Flexibility.** While a control algorithm can be shown to be optimal under a set of restricted operating conditions, a useful control algorithm must have bounded performance over a wide range of operating conditions.
- **Robustness.** Hardware failures are inevitable in any large system. A good control algorithm should be robust in the face of hardware failures; it would be preferred if the algorithm were smart enough to move data out of failing nodes.
- **Retries.** In the Q-Machine implementation, blocked messages are resent; each delivery attempt increases the effective latency of the message by an amount proportional to the resend backoff time. An intelligent migration control algorithm should recognize these situations and abort the migration process if the additional latency of resending the migration packet will hurt overall performance.

7.2.2 Languages and Compilers

ADAM was developed in parallel with languages that could leverage its unique features; in fact, over the course of development, two languages, Couatl and People, were developed by Ben Vandiver [Van02].

Couatl is the first language developed for the ADAM platform; it is a simple object-oriented language that employs a technique known as *persistent methods* to perform object dispatch. Persistent methods are started once for every instance of an object, and never terminate. Every object has associated with it a persistent method which acts as a server for method invocations. The persistent server method waits on a client to enqueue a method invocation request into a designated request queue. Upon receipt of a request, the server method performs a method lookup and spawns a new thread of execution for

that method. Couatl was primarily developed to prove that the ADAM programming model is viable and that reasonable compiler analyses can generate code that makes use of the queues without deadlocking. It also proved that the thread-per-method model is a viable programming model. The primary problems with Couatl include no programmer-level visibility of the queue structures, and no inherent support for spatial awareness. Code generated with Couatl would place all methods and new objects onto a single node, and relied on the load balancing mechanism to improve performance.

People (from PPL, Parallel Programming Language) is the successor to Couatl. People also sports an object-oriented programming model. Its most significant addition is support for streaming constructs that expose the queues within the core of the machine to the programmer. Streams represent a way for a programmer to explicitly schedule static communication patterns. These streaming constructs were used in the Matrix Multiply benchmark, for example, to set up a static array-access/multiply pipeline. They were also used in the N-Body benchmark to create a static combining tree for determining when all threads were finished computing their result during each time step.

Future languages for the ADAM architecture should also include primitives to stripe, split and scatter arrays and vectors. A parallel-map operator would also be useful, as well as mechanisms to simplify the building of fan-in and fan-out trees. Also required is run-time support for garbage collection. For an efficient implementation of parallel garbage collection, please refer to Jeremy Brown's thesis on Sparsely Faceted Arrays (SFAs) and scalable parallel garbage collection. [Bro02]

7.2.3 Hardware Implementation

An important step in any architecture's evolution is its hardware implementation. Fortunately, the very nature of an abstract machine architecture lends

itself to incremental implementation. In addition, Q-Machine implementation's dilation-2 fat-tree network has fault tolerance built in; [DeH93] describes in detail how the network implementation can withstand a single failure in any component or link without any loss of logical connectivity. Finally, the ADAM architecture and the Q-Machine implementation were designed with the practical issues of manufacturing yield and obsolescence resistance in mind. Manufacturing yield on a Q-Machine chip-multiprocessor die can be near 100%. Thanks to the hardware abstraction of the ADAM, chips with multiple bad processors are still salable; the runtime system just needs a map of the bad locations so that no data or threads are migrated into the broken nodes. The hardware abstraction of the ADAM also helps extend the operational life of the architecture; as nodes malfunction, they can be replaced with new nodes implemented in the latest process technology without a need to recompile. Combined with a migration control system that can move data and threads out of a failing node, a system could run for years while being constantly upgraded – without ever being shut down.

7.2.4 Transactions

Hardware support for transactions is very useful in large, parallel architectures. Transactional rollback enables greater levels of speculative parallelism, and transactional checkpointing enables a greater level of dynamic fault tolerance. The ADAM architecture has some unique features that enable future implementations of transactions in hardware.

The first observation is that a queue can be turned into a transactional log if the "dequeue" operation is reversible. In other words, a dequeue operator should merely advance a *dequeue pointer*, without throwing away any data. Given this transformation, the computational state of an ADAM thread can be saved by simply remembering all of the enqueue and dequeue pointer offsets.

In order to reclaim memory, computation can be committed after the necessary conditions have passed by throwing away some data. Memory state can also be preserved with this scheme by using only exchange operators on memory, and reversing the exchanges during rollback.

The second observation is that an ADAM thread's state is entirely represented within the thread capability. Hence, checkpointing can be done at a coarser grain than the previously suggested pointer-rollback method by just making copies of thread state. This coarse-grained transaction mechanism is easier to implement, and requires less hardware modification.

Of course, the devil is in the details. Many issues, such as how to deal with migration and non-deterministic message ordering between threads, need to be resolved before either scheme can be declared a success.

7.3 Final Remarks

As the Future Work section indicates, the ADAM architecture and the Q-Machine implementation are full of low-hanging fruit. In addition, the ADAM architecture has implications for high performance parallel computers beyond just enabling a high performance data and thread migration scheme. I hope to explore these possibilities in the near future. I also encourage anyone who has taken the time out to read my thesis to investigate and to implement aspects of the architecture, and to please feel free to send me an email if they have any questions. My email address for life is `bunnie@alum.mit.edu`. I look forward to hearing from you.

Appendix A

Acronyms

> Q: What do you think will be the biggest problem in computing in the 90's? A: There are only 17,000 three-letter acronyms.
>
> —*Paul Boutin from* The New Hacker's Dictionary

This chapter lists, for the reader's convenience, the acronyms and abbreviations used in this thesis.

$	Shorthand for cache
ACK	Shorthand for Acknowledge
ADAM	Aries Decentralized Abstract Machine
AM	Attraction Memory
AMD	Advanced Micro Devices
ASIC	Application Specific Integrated Circuit
ASS	ADAM System Simulator
BIST	Built In Self Test
ccNUMA	Cache-Coherent Non Uniform Memory Access
CM	Connection Machine
CMOS	Complementary Metal Oxide Semiconductor
CMP	Chip Multi-Processor *or* Chemical-Mechanical Polishing
COMA	Cache Only Memory Architecture
Couatl	Java-derivative object-oriented proto-language for ADAM
CPU	Central Processing Unit
CTO	Chief Technology Officer
DAE	Decoupled Access Execute
DDM	Data Diffusion Machine
DMA	Direct Memory Access
DMEM	Data Memory
DRAM	Dynamic Random Access Memory
ECC	Error-Correcting Code
EMEM	Environment Cache
EXEC	Execution Unit of the Q-Machine core
FOn	Fan Out of n
I$	Instruction Cache, often abused to refer to a hybrid work-window scheduler queue
IBM	International Business Machine
ID	Shorthand for Identifier
IP	Instruction Pointer *or* Intellectual Property
IPC	Instructions Per Clock
ISA	Instruction Set Architecture
KSR	Kendall Square Research
LSB	Least Significant Bit
MIMD	Multiple Instruction Multiple Data
MIT	Massachusetts Institute of Technology
MSB	Most Significant Bit

Table A.1: Table of Acronyms

NI	Network Interface
NOW	Network of Workstations
NSRF	Named State Register File
NUMA	Non-Uniform Memory Access
ORB	Object Request Broker
OSI-7	Open Systems Interconnection 7-layer model
PC	Program Counter *or* Personal Computer
People	Second-generation language for ADAM, supporting streaming constructs
PHY	Physical and Data Link Layers (from OSI-7 model)
PIM	Processor In Memory
PPL	Parallel Programming Language (a.k.a. People)
PQF	Physical Queue File
Q$ *or* QC	Queue Cache
RPC	Remote Procedure Call
RISC	Reduced Instruction Set Computer
SCHED	Scheduler Co-processor
SFA	Sparsely Faceted Array
SGI	Silicon Graphics Incorporated
SIA	Semiconductor Industry Association
SMEM	Scheduler Co-processor Memory
SMT	Simultaneous Multithreading *or* Surface Mount Technology
SOI	Silicon on Insulator
SRAM	Static Random Access Memory
SSRAM	Synchronous SRAM
src	abbreviation for Source
TAM	Threaded Abstract Machine
TSMC	Taiwan Semiconductor Manufacturing Corporation
TTDA	Tagged-Token Dataflow Architecture
VLIW	Very Long Instruction Word
VQF	Virtual Queue File
VQN	Virtual Queue Number
XPRT	Network and Transport Layers (from OSI-7 model)

Table A.2: Table of Acronyms, continued

Appendix B

ADAM Details

> My two favorite languages are still assembly and solder.
>
> —*bunnie*

This appendix provides many of the details omitted from chapter 3 in the interest of restricting the body text of the thesis to only features and issues relevant to migration. Note that this appendix does not justify all the design decisions as rigorously as the main chapters of the thesis; in fact, some of the implementation decisions, such as the use of a simplified floating point format, are mostly a result of my own prejudices. On the other hand, the choice of floating point format has little to do with the meat of the architecture, and I present such material here just because I can.

B.1 Data Types

All ADAM data types are 80 bits wide; they consist of a 64 bit data field and a 16 bit tag field. Four integer data types are supported: signed long (referred

to as “word”), packed signed integer, packed signed short, and packed unicode characters. Only one floating point data type is supported, similar to the IEEE-754 double format. See figure B.1 for detailed bit-level formatting of the data types.

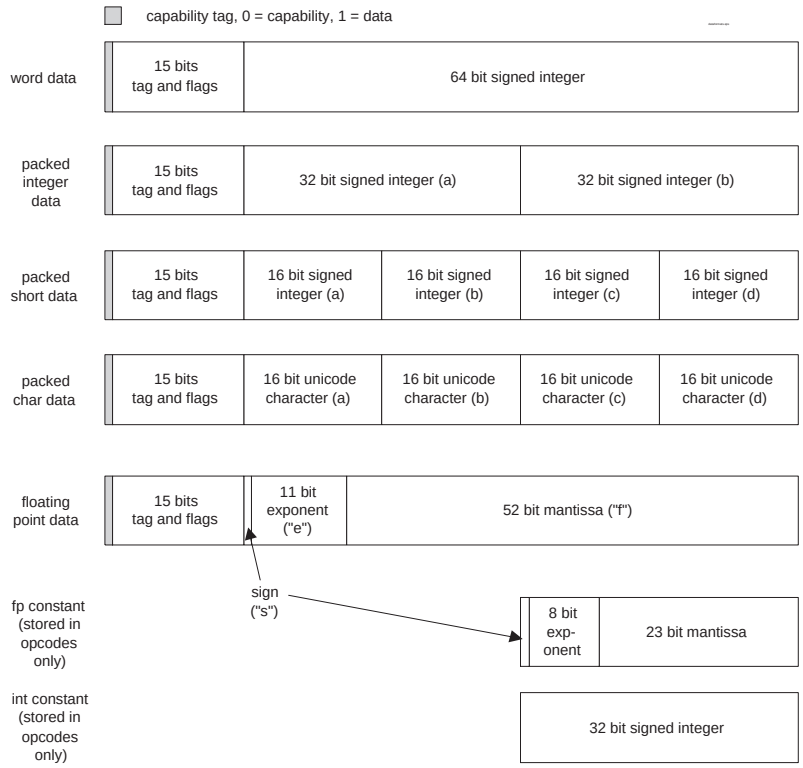


Figure B.1: Data formats supported by ADAM

Packed data is operated on in vector form; most arithmetic operations are supported on packed data. Any arithmetic operation involving a capability, however, is only valid with a word. Any integer type is supported for memory queue offsets, however. Please see section 3.2.3 for more information on the ADAM memory model.

All data types are fully tagged to identify their type, as well as any flags associated with their status. See figure B.2 for details. Errors on arithmetic operations can be forced to be trapping and non-trapping. Trapping errors cause the thread to halt and an exception to be thrown; non-trapping errors allow execution to proceed normally (which may or may not imply halting) and the error condition to simply be noted in the result's tag and flags field. This error condition will propagate through data operations; in other words, adding a NaN-tagged float with a valid float will result in a NaN-tagged float.

An immutable bit is included in the tags to indicate static data that cannot be altered. Identifying data as static allows management routines to copy immutable data freely, thus enabling cheap automatic mechanisms for distributing frequently referenced constants. Writing to data that is declared as immutable has no effect on the data, may throw an exception, and always sets a bit in the status register to indicate that an illegal write occurred.

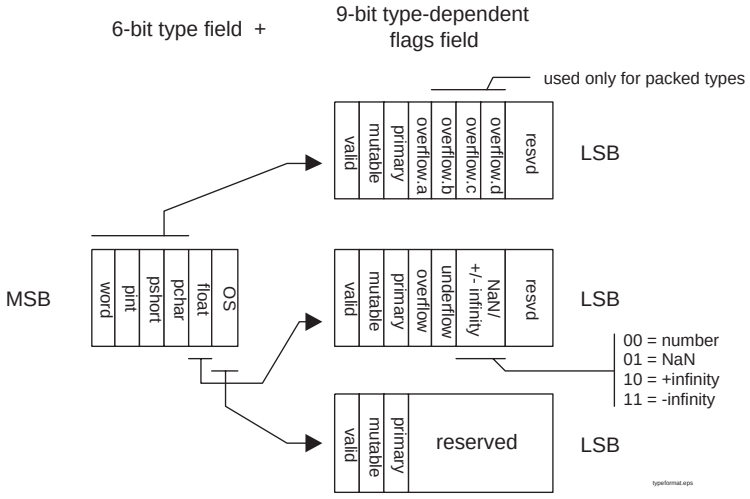


Figure B.2: Tag and Flag field details

A primary bit is also included in each tag that is used by the data mi-

gration manager to indicate if this is the primary copy of the data. This is particularly useful for the scenario of partial migration, where the primary capability containing some data has migrated but the data itself has yet to move. See chapter 4 for more details on migration mechanisms and implementation.

A subset of the IEEE 754-1985 floating point standard is required by ADAM architecture. The differences between the IEEE 754-1985 standard and the ADAM format are chosen to simplify implementation and enhance performance with a small reduction in precision. These differences are:

- ADAM does not support single-precision floats and its associated operations and conversions, with the exception of constant fields in opcodes
- NaN and $\pm\infty$ are specified in the tag field, so exponent = 2047 is now valid, and the exponent bias is now +1024
- ADAM has no denorms (accuracy versus IEEE 754-1985 reclaimed by indicating special number types in the tag field, as described immediately above)
- one rounding mode: von Neumann style rounding

To summarize, the value of the floating point number is $v = (-1)^s 2^{e-1024}(1.f)$ unless $e = 0$ and $f = 0$, in which case the value is $v = (-1)^s 0$ (signed zero).

Aside from these differences, the ADAM floating point format defers to the IEEE 754-1985 standard [Ste85]. In particular, the handling of NaNs, Infinity, and Signed Zero in the context of Exceptions, Traps, Comparisons and Conversions are identical.

The ADAM instruction format allows for 32-bit constants to be stored in a standard opcode. Floating point instructions can thus store a single-precision format float in the constant field, but this is immediately converted to a double-precision number upon use. The single precision floats likewise do away with the denorm representation; hence, NaNs and $\pm\infty$ are not rep-

resentable in the single-precision floating point constant field. The value of a single precision floating point number is $v = (-1)^s 2^{e-128}(1.f)$ unless $e = 0$ and $f = 0$, in which case the value is $v = (-1)^s 0$ (signed zero).

von Neumann style rounding is implemented by adding a Least Significant Bit (LSB) of precision to floats as the floats enter the arithmetic pipeline, and carrying this LSB of precision throughout the pipe. This extra LSB is set to a binary "1" as numbers enter the pipe, and rounding is done by simple truncation at the end of the pipe. This results in an expected value of the extra LSB to be $\frac{1}{2}$ at the end of the day.

An implementation may choose use to full IEEE 754-1985 style rounding to gain the extra precision, but there is no provision in the stock architecture specification to choose which rounding mode to use; the default and only rounding mode should thus be "round to nearest" per IEEE 754-1985.

B.2 Instruction Formats

ADAM has a sequestered code space, like that in a Harvard Architecture. The code space, unlike the data and environment spaces, is global and shared among all nodes; this is feasible because the code space is *mostly* read-only. The management coprocessor takes care of handling any page faults or the loading and unloading of code in code space. ADAM can dynamically request new object classes to be loaded into code space with the `LDCODE` instruction.

The code space is mostly read-only because some instructions contain hint fields to the instruction prefetcher. The actual values contained in the hint fields are implementation-dependent and any ADAM implementation must execute code correctly regardless of the hint field's contents; however, a compiler is free to warm up the hint fields with bit patterns that may improve

start-up performance for a specific implementation. Instruction caches can replace lines that have not been written back due to a lack of instruction memory bandwidth without any impact on correctness of execution. Likewise, write values do not have to propagate throughout the system even though the code is globally shared among all nodes. However, in the case that values do make their way back to their original file on disk, the next time code is loaded, it may run faster.

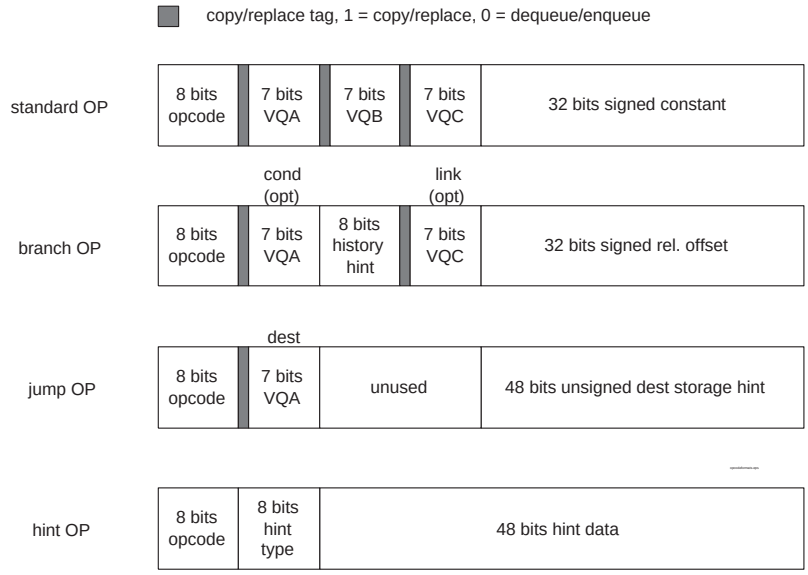


Figure B.3: Format of ADAM opcodes

Instructions are 64 bits long and have four basic formats: standard, branch, jump, and hint (see B.3). Every instruction has an 8-bit opcode field. Every queue specifier in every instruction is modified by a copy/clobber bit. Setting the copy/clobber tag enables the compiler to treat the queue with semantics similar to that of a register. A copy operation extracts a value from a queue without changing any of the values in the queue; a clobber operation tests to see if a queue is empty, and if it is, waits until a value is written to it, and then

replaces the value. The clobber operation is invalid on a remapped queue and attempting to perform such an operation triggers an exception.

The standard instruction has three virtual queue specifiers, each 7 bits long. The first two (VQA and VQB) specify read queues; the final (VQC) specifies the write queue. The standard instruction also contains a 32-bit signed constant field, thus allowing the standard instruction to specify up to three data sources and one data destination, although most instructions do not take advantage of this possibility.

Certain instructions, known as special-format instructions, may interpret the VQA, VQB, or VQC fields as constants instead of as a queue to reference to extract or store data to the queue file. These instructions typically deal with the creation, maintenance and destruction of queue maps. The compiler and/or assembly language programmer typically knows at all times the exact queue number that a mapping is applied to, so it does not make sense for most queue map maintenance instructions to accept arbitrary dynamically generated queue values. Hence, the VQA, VQB, and VQC fields can be used to immediately refer to a queue number for these instructions.

Branch instructions have a condition field, a link field, a branch history hint field, and a 32-bit signed branch offset. Either the condition or the link field may be omitted from an instruction, but not both. An 8-bit history hint field is also provided so that a branch history can be stored with the branch instruction. Note that the format of the hint field is implementation-specific, and that any ADAM implementation must function correctly regardless of the hint field contents.

Jump instructions have a destination field and a 32 bit unsigned jump destination hint. Only the lower 32 bits of the value in the queue specified by the jump destination field is loaded into the program counter. The jump destination hint field is provided so that an implementation can memoize the most

recent jump address. Note that the format of the hint field is implementation-specific, and that any ADAM implementation must function correctly regardless of the hint field contents.

Hint instructions are no-ops that provide hints to the runtime system. The hint may or may not be platform dependent; this information is encoded within the hint type field. Examples of hints are data placement directives, prefetch directives, and thread yield directives. Hints that are not recognized by the run-time are ignored.

Please consult appendix D for detailed listing of the instructions supported by ADAM and their descriptions.

B.3 Capability Format

The capability format used by ADAM [BGKH00] allows for exact base and bounds determination from an arbitrary capability with the use of front-padding to eliminate a small amount of rounding overhead. The total padding penalty incurred by the capability format is bounded to be less than 11.2% [BGKH00].

The method for extracting the base and bounds from a front-padded capability is fairly simple, and can be implemented directly in hardware. As seen in figure B.4, a capability includes block size, length, finger and address fields. A combination of block size and length can be used to determine the end of a capability; the finger field is used to deduce the location of the beginning of the capability given a pointer into the middle of the capability. A block size of all 1's (63 in this case, because the block size field is 6 bits long) is a special case where the length field is directly equal to the number of words in the capability. This unique structure was chosen to simplify the hardware implementation, as described in [BGKH00].

The method for extracting the base and bounds from a front-padded ca-

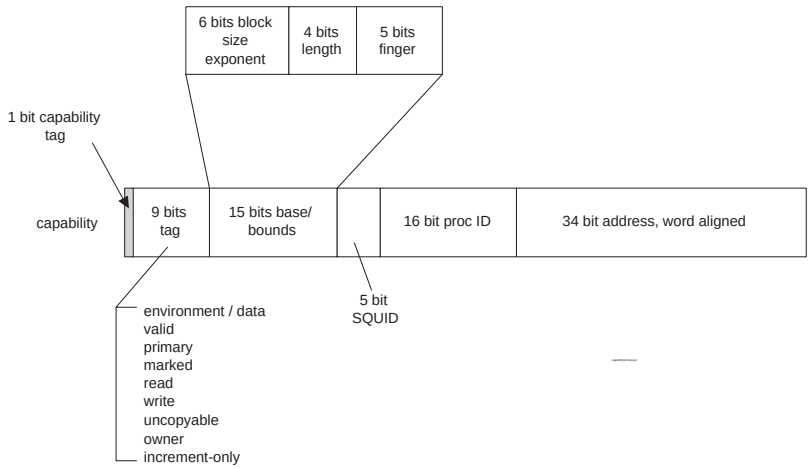


Figure B.4: ADAM capability format

pability is as follows, written is pseudocode:

```
B = block size field value
L = length field value
F = finger field value
A = address field value

if( B == 63 ) {
   // L, B are immutable
   // A and F are updated by capability arithmetic ops,
   // with check made to ensure that F < L
   capability.beginning = A − F;
   capability.length = L + 1;
   capability.end = capability.beginning + capability.length;
   if( F ≥ L ) {
      throw capability bounds exception
   }
   desired data = *A;
}
else {
   // & is the bitwise AND operator
   capability.beginning = A & (∼ ((1 ≪ B) − 1)) − (F ∗ (1 ≪ B));
   capability.length = (1 ≪ B) ∗ (L + 1);
   capability.end = capability.beginning + capability.length;
   desired data = *A;
```

```
}
```

The only valid operations on a capability are addition and subtraction. The new address that results from an arithmetic operation is simple to calculate:

```
X = signed integer offset to be added
A2 = new address
A1 = old address

A2 = A1 + X
```

The method for recalculating the finger field of a capability that has had an arithmetic operation on it is as follows, written in pseudo-code with verilog bitfield syntax:

```
F2 = original finger field
F1 = new finger field
X = signed integer offset to be added

B = value of the block length field
if( B == 63 ) {
   F2 = F + offset;
} else {
   F2 = (A + offset − ( A & (∼ ((1 ≪ B) − 1)) − (F ∗ (1 ≪ B)))) ≪ B;
}
```

The value of the new finger field should be less than the value of the length field but greater than zero; if not, an error should be flagged. An efficient hardware implementation of the above calculation is also given in [BGKH00]. Note that capabilities cannot be dynamically resized. This implies that the length and block size fields should never change after an arithmetic operation. In order to grow a capability, a new one must be created and the contents of the old one copied into the new one.

The ADAM capability format contains an explicit processor node ID embedded within the address field of the capability. The size of the node ID field allows for up to 65,536 processors to be present in the system, but the actual allocation of capabilities on these nodes is left up to the operating system.

All ADAM applications can run on implementations with anywhere between one and 65,536 nodes, with no requirement on the distribution of node IDs, because capabilities are opaque to the programmer and the allocation process is implementation-specific. Valid node IDs can even change dynamically, so long as the OS is careful to ensure that a node is empty before deactivating its ID. Dynamic ID reassignment can be useful in situations where environmental monitors detect an impending failure, or where users wish to hot-swap nodes to perform upgrades or service. Note that the amount of available memory for applications to run does vary with the number of nodes in the system, but the address space is fairly large so users should rarely encounter this situation.

The capability format also includes a number of bits for memory management and security purposes. These bits are:

- **environment/data**: indicates if the capability is for environment space or for data space. Normally this bit should not be modified after capability creation.
- **increment-only**: indicates that only positive offsets from the capability base can be accessed
- **valid**: indicates if a capability is valid. An attempt to dereference an invalid capability results in a protection fault.
- **marked**: used for garbage collection
- **read**: indicates that data can be read from the capability.
- **write**: indicates that data can be written to the capability.
- **uncopyable**: indicates that only dequeue operations are allowed on the capability; an attempt to copy the capability will result in an exception being raised.
- **owner**: when the owner bit is set, the read, write, and uncopyable bits can be overridden.

- **primary**: indicates that this capability is the primary working copy. For capabilities in data space, it marks the endpoint of a migration list. For capabilities in environment space, it also marks a thread with this bit set as the only runnable copy.
- **SQUID**: **S**hort **Q**uasi-**U**nique **ID**. A short tag field that contains a randomly generated ID number assigned at the time of capability allocation; when a capability is migrated, this field is directly copied. Use of this field reduces the cost of capability inequality comparisons. [GBHK00]

B.4 Über-Capability and Multitasking

The über-capability is a capability that has access to the entire memory space of the machine. This über-capability is used by kernel threads for system management functions, since ADAM provides no supervisor mode or explicit kernel permissions in the style of Java. On power-up, each physical node starts code execution at location 0 in code space, and an über-capability is initially placed in q0. The über-capability is set to be the size of the entire virtual memory available for that node, and the owner bit is set. Through this mechanism, kernel code loaded at location 0 in code space can have access to the entire machine. This kernel code is also typically the default exception handler for the node.

Since ADAM is a virtual machine, multitasking on a single large machine is accomplished by dividing the machine into smaller groups of physical nodes and starting an ADAM per task, and each ADAM runs only one task. Load balance of the machine can be set in part by controlling the number of nodes that an ADAM can access. This restriction of access can be accomplished in part by limiting the size of the über-capability.

B.5 Exception Handling

Exceptions on ADAM are inherently imprecise. ADAM is a distributed machine that runs many parallel threads; there is no clear definition of "simultaneity" in this scenario. ADAM's take on exceptions is two-pronged: first, the result of every exception-causing event is tagged; second, as much local state relevant to the exception is preserved at the instant an exception is detected.

An exception capability is included as part of every thread's state. This exception capability is initialized on thread creation to point to a default exception handling object (usually an OS-defined object), and can be overridden by the user at any time. The default handler is invoked in the case that the user-defined exception handler is invalid. Users can use this mechanism to build chains of exception handlers, as illustrated in figure B.5.

Exceptions are handled on a per-processor node basis. When a thread encounters an exception, the exception handler is immediately scheduled to run on that node, and is locked in as the only runnable thread until the exception is resolved. The exception handler inspects the processor status register and the Exceptioned Context ID register to determine the source of the exception. Then, an OS-defined protocol is employed for communicating with the exceptioned thread, if necessary. Usually, this protocol involves the exception handler forcing a flush of the exceptioned context from the queue file and digging through and modifying the exceptioned context's environment space. This process can take thousands of cycles. Exceptions are intended to be rare events, and users should avoid using the exception mechanism for anything other than exception handling. In other words, they should be avoided in general as a mechanism for implementing APIs or hardware interfaces. Users are instead encouraged to use queue mappings and opcode extensions of a

form similar to ALLOCATE or SPAWN instructions. Opcode extensions can be implemented using the illegal opcode handler mechanism described below.

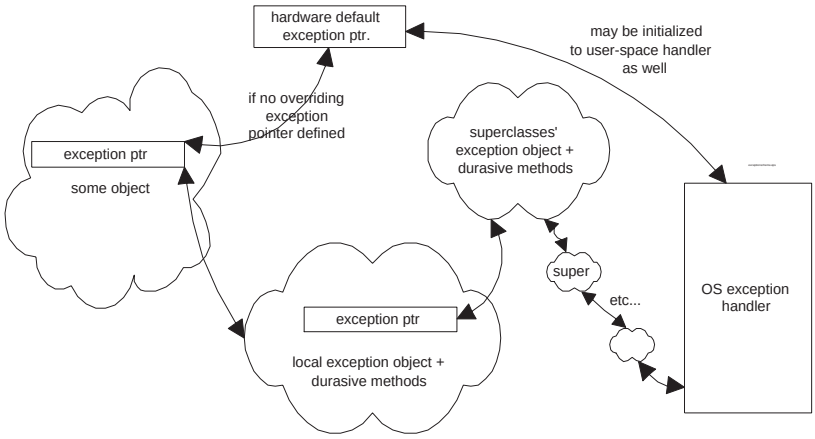


Figure B.5: Exception handling overview

Illegal opcode exceptions are handled in a special manner, similar to the Alpha architecture's PALcode. An illegal opcode dispatches into a look-up table in memory that has a hard-wired address, and control flow is transferred to an implementation-specific microcode processor that has access to all local state. The microcode processor could be as simple as a dedicated context ID on the ADAM plus instruction set extensions. The code that the microcode processor executes is stored in a reserved location in kernel memory; this allows for instructions implemented in future versions of the architecture to be emulated via software patches set up by the OS. During emulation mode, the processor behaves as if it had stalled, and errors during emulation mode lead to undefined behavior. I recommended that the default behavior for an illegal opcode be an emulated THROW instruction.

Appendix C

Q-Machine Details

> A novice was trying to fix a broken Lisp machine by turning the power off and on.
>
> Knight, seeing what the student was doing, spoke sternly: "You cannot fix a machine by just power-cycling it with no understanding of what is going wrong."
>
> Knight turned the machine off and on.
>
> The machine worked.
>
> *—Traditional AI Koan*

This appendix provides many of the details omitted from chapter 5 in the interest of restricting the body text of the thesis to only features and issues relevant to migration. In particular, this section discusses the details of the PQF implementation, the network interface and transport protocol implementation, and the network topology used in the Q-Machine implementation and the ADAM System Simulator.

C.1 Queue File Implementation Details

This section discusses some of the important details of the PQF implementation used by the ADAM system simulator. This piece of hardware is perhaps the most difficult single component to implement in the Q-Machine implementation, so it warrants some exploration within the context of this thesis.

C.1.1 Physical Design

At the heart of the VQF is the physical queue file (PQF), which directly implements an architecturally unspecified number of queues. A high-level sketch of a PQF can be seen in figure C.1. The PQF is attached directly to the computational units. The size of the PQF should be set by the details of the target implementation process; however, for good single-threaded performance, the PQF should embody at least the 128 queues available to a single context. The PQF has a structure similar to a multi-ported register file, and it is capable of swapping an entire queue into and out of a Queue-Cache (QC) in a single cycle. Empty queues are not swapped into the QC; rather, they are simply marked as empty and they consume no further bandwidth or space. The memory subsystem contains special hardware to accelerate the marking and swapping of empty queues. A good compiler will arrange for threads to have all empty queues when execution stops, so that dead threads consume a minimal amount of space until they are garbage collected.

The QC has a structure similar to a memory cache; when it overflows, cache lines are strategically written out to main memory. The fact that every queue in the system has some location in memory reserved for its storage is a feature that is used by the GC mechanism to clean up after dead threads or to migrate objects.

The physical queue file actually does not take up significantly more space

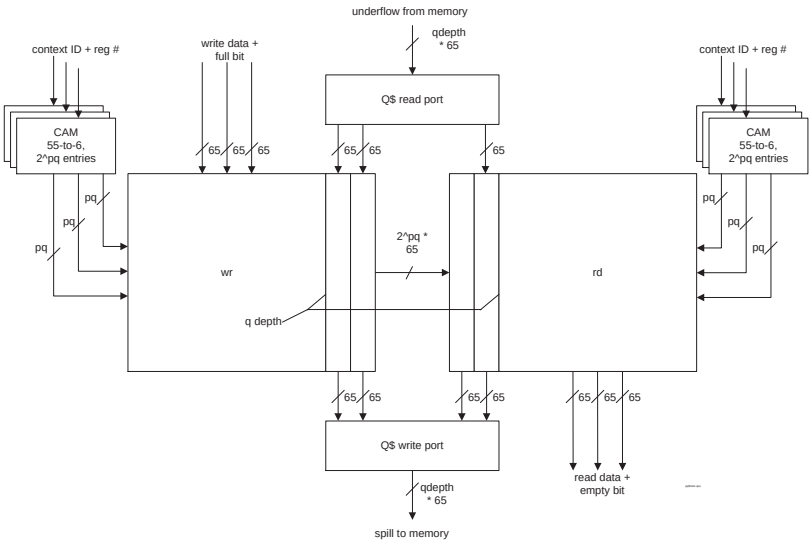


Figure C.1: A 3-write, 3-read port VQF implementation. pq = $\log_2$(# physical registers). Q-cache details omitted for clarity.

than a regular multiported register file. The reason for this is the fact that a register file is wire-dominated; the active transistor area underneath a register file cell is a small fraction of the area allocated for wires.

Figure C.2 illustrates the unit cell for a 3 read-, 3 write-port PQF with sufficient Q-cache wires to manage a 4-deep queue.

The wiring pitch is based on numbers taken from the TSMC 0.18 μm process guide [Corb]. The wiring requirements for the unit cell of the PQF would consume 4851 λ^2 alone, using minimum-pitch M5/M6 wires. For comparison, the area of a 6-T SRAM cell in the TSMC 0.18 μm process is 574 λ^2, allowing eight such cells to be placed underneath a PQF unit cell. For better performance, fatter wires with wider spacing may be employed, thus increasing the area underneath the unit cell for the implementation of the actual Q structure storage and control logic.

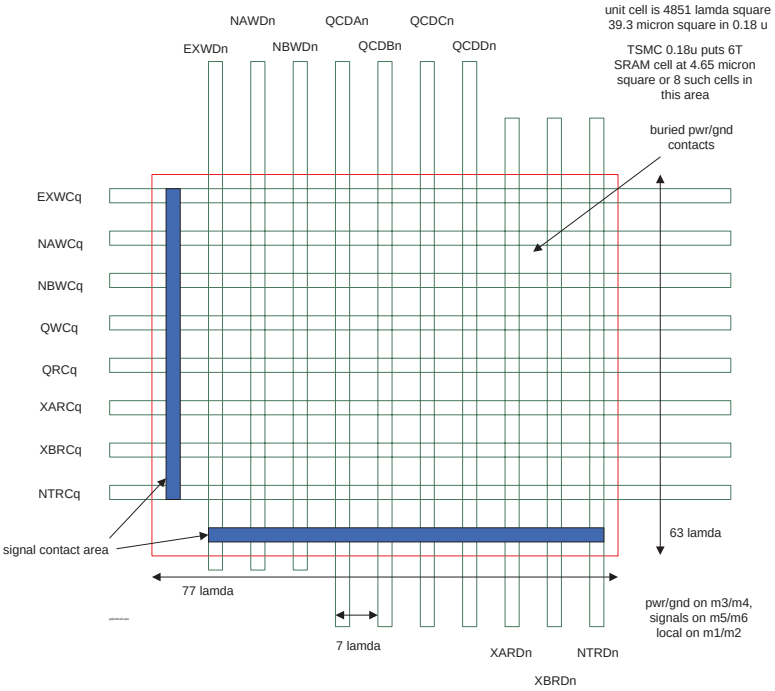


Figure C.2: PQF unit cell.

Hence, a PQF implementation which has relatively shallow queues (4 to 8-deep) could be implemented within a factor of two of the amount of space as a regular register file with a similar number of ports. As process technology progresses, even greater depth queues will be enabled, at the expense of either more or faster wires required for swapping to the Q-cache. A suitable, high-performance asynchronous FIFO design is described in [MJC+99] and [MJCL97]. These depth-17 FIFOs operated reliably at a throughput of 1.7 Giga data items per second in a 0.6 μm CMOS process. Variants on this design have been explored by the author but are not presented here in the interest of brevity.

A similar idea to the VQF implementation outlined here is the Named-State Register File (NSRF). [ND91] [ND95] The NSRF is a register file with an automated mechanism for spilling and filling thread contexts. It utilizes context ID numbers to uniquely identify the threads, and a CAM memory to match the individual register file entries to their proper contexts. Unlike the VQF, the NSRF dumps its state directly into the processor data cache. The Q-Machine does not do this because there is no data cache on the Q-Machine, and even if there were, the combination of having to add an extra read/write port to the D-Cache and cache pollution issues would present a strong case for having a separate Q-cache. While the VQF is introduced primarily to support a disassociated physical-to-logical mapping of processors to threads, it is interesting to note that the NSRF did provide small (9% to 17%) speedups to parallel and sequential program execution. Also of note is that cache-style register files such as the NSRF and VQF provide higher overall register file utilization: the NSRF was demonstrated to have 30% to 200% better utilization than a conventional register file. [ND95]

The MAP block is responsible for determining if a queue is mapped to another context. The MAP block is issued a request to discover a queue

mapping at the time the instruction is issued, giving it the whole pipeline latency of the machine to do this work. The MAP operation is potentially complex and could be a cause of many stalls if the machine is not designed correctly.

The reason the MAP block only needs to be decoded for write targets is because the only legal queue mappings allowed on the Q-Machine are forward mappings. In other words, it is impossible to create a mapping that ”pulls” data out of another context; instead, one can only inject data into a target context. As apparent from the diagram, the MAP function is thus invoked for both incoming writes from the NI and for local results from the ALU/MEM unit. This keeps the read latency from the VQF low, while giving the MAP function time to do its translation for writes.

Recall that the context ID for a thread is in fact a capability that points to the storage region for the thread's backing storage and local data storage. This capability has permissions set such that a user process cannot dereference this capability and use it as a memory pointer, but the OS and MAP function have access at all times to this information. Refer to 3.2.3 for a review of the capability address format of the Q-Machine. Given this, the basic algorithm for the MAP block is as follows:

- If the Proc ID field of the context ID does not equal to the Proc ID of the local processor, send the write to the NI
- Otherwise, consult an internal cache that records the presence of a mapping on the specified queue for the specified context. If there is no map present, pass the write on to the VQF. If there is a map present, consult the map table to discover the proper mapping and ship the data off to the NI for routing (even if it is a map-to-self). Mark the queue as full and block the thread until the NI reports successful delivery of data

  The map presence cache is used to help accelerate the typical case where

there is no mapping. A larger map presence cache can be held in memory than a cache with presence bits and the actual mappings. In the case that the mapping table overflows, a lookup into a backup table must occur and the machine thrashes. Also, in the case that a mapping does exist, it is okay to take a few extra cycles to retrieve the mapping from memory. Perhaps a small cache of mappings will also be maintained if the mapping lookups are determined to be a severe bottleneck.

C.1.2 State Machine

Fill requests from the PQF are generated in response to both missed read and write requests by issued instructions. For read requests, a placeholder line is marked in the PQF and the fill request is issued to the environment memory. For write requests, it is more complicated. If the queue was never created before in the context, an empty line in the PQF is simply converted to a full-fledged read/write line with the dirty bit marked. If there is an existing read placeholder, that line is converted into a write-only line with the write data, pending a merge fill with the already issued fill request. Otherwise, if there is space in the PQF, a write-only line is created and the merge read fill request is issued. If there is no space in the PQF for the write request, a request for an empty line is made, and once that is satisfied, a write-only line is created with a merge fill request.

Flush requests from the PQF are generated in response to the following events: PQF overflow, scheduler overflow, and migration. In the case of a PQF overflow, the LRU line is chose and booted out of the queue file. The following two cases are not documented in figures C.3 and C.4. In the case of scheduler overflow, the scheduler can no longer track all the ancillary data associated with a context and it wishes to retire the entire thing to environment memory. In the case of a migration request, the entire thread state must

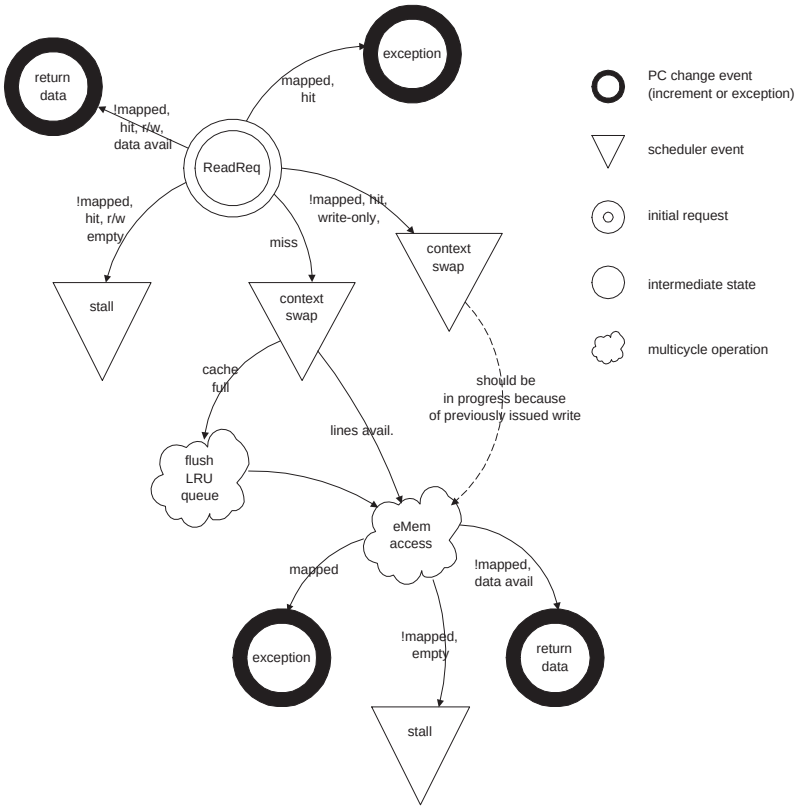


Figure C.3: PQF read request response flowchart

once again be retired to memory, but in addition some data may be forwarded via the network interface before the retirement is finished. In order to get write-only lines with merge-fills and dirty lines to be handled correctly on the destination of the migration, a migrated line must move with its tags and inserted into the destination queue along with any pending requests associated with the tag type. Eventually, the fill mechanism will work its way to get the data from the original context via forwarding pointers, but in the meantime, computation can resume. Any fills in progress to remedy placeholder or write-only lines locally are allowed to complete, but the returned data is discarded. This is acceptable because write-only lines are retired to environment memory with a merge request (and as previously noted, write-only lines directly sent to the destination are inserted with merge requests). Placeholders, of course, can simply be discarded.

The following state is also stored in a PQF line or must be synchronized with the environment memory backing storage upon retirement, in addition to the raw queue data:

- Created bits
- Resident bits
- Mapped bits
- Map destination context
- Map destination VQN
- Map source queue flag (also impacts network interface)
- Map source queue sister (also impacts network interface)

The map source queue flag and map source queue sister values must be reported to the network interface whenever a queue is swapped in or out of the PQF. This is because the originating thread data is stripped from an incoming network packet by the transport layer implementation. Thus, when the map source queue flag bit is set, the transport layer must preserve the originating

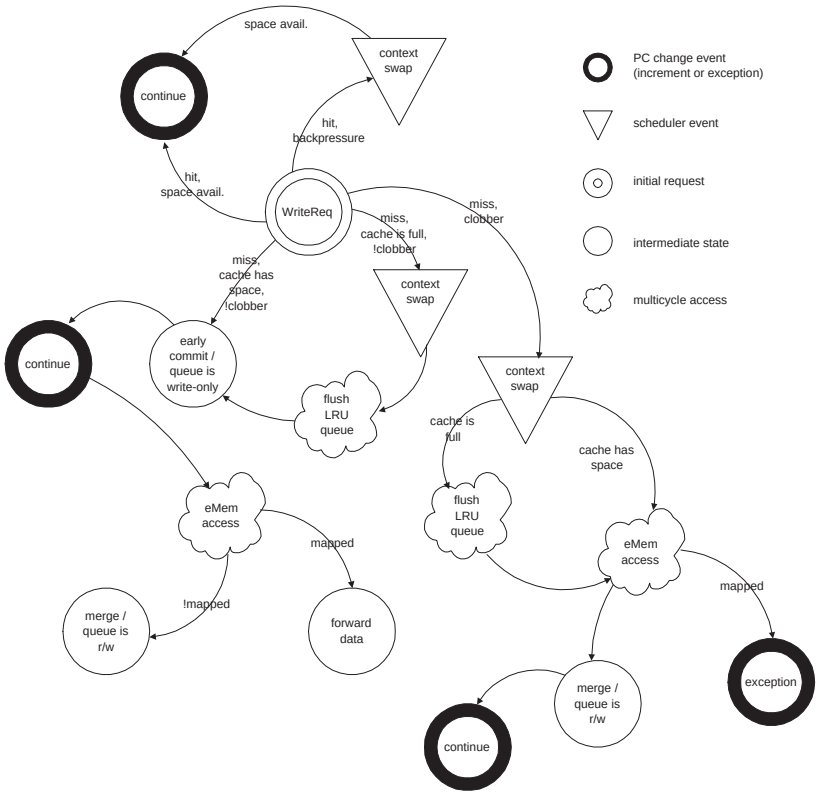


Figure C.4: PQF write request response flowchart

thread context from the network packet and generate a write request into the PQF for both the arriving data and the context ID of the originator of that data.

C.2 Network Interface

The network interface implements, in hardware, features analogous to the physical, data link, network, and transport layers from the OSI 7-layer network stack. In this implementation, the physical and data link layers are combined and referred to as PHY, and the network and transport layers are combined and referred to as XPORT. This reduction of abstraction was chosen because first, the network protocol for the ADAM implementation is very simple, and second, there is a strong motivation to reduce latency by cutting out unnecessary buffering and packet encapsulation. That being said, the network interface provides the following services:

- an abstract and modular interface from the processor core to the physical network layer
- reliable delivery of data
- generation of stalls to the processor core when the network is congested
- idempotent delivery of data
- a cut-through path to memory nodes that bypasses XPORT- and PHY-layer latencies
- in-order delivery of data
- a loopback path for inter-thread data on the same processor node

The network interface assumes that the PHY has no responsibility for packet delivery and no packet buffering. Hence, the XPORT layer must implement reliable delivery and attempt to always guarantee space for arriving packets. A discussion of how the topology and routing in the PHY layer is

implemented can be found in section C.3. Readers are recommended to consult that section if they are unfamiliar with circuit-switched worm-hole routed networks, and unreliable (but fast) routing.

An overview of the network interface implementation can be found in figure C.5. Data coming from the processor core first has its source and destination headers appended, and then routed by cut-through and loopback routers. The datapath at this point routes source, destination and data information in parallel to reduce latency.

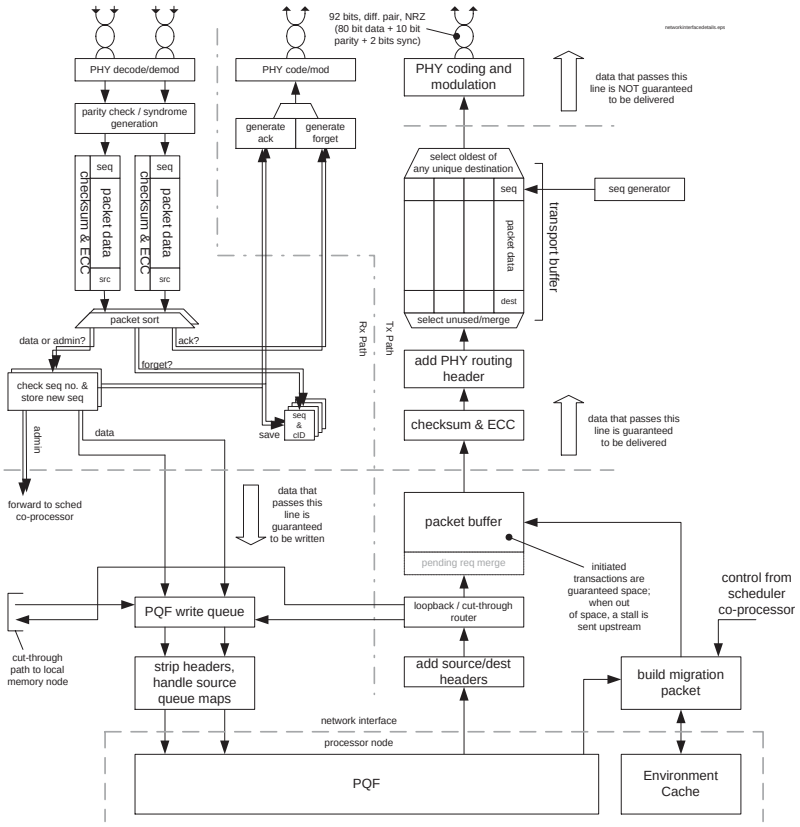


Figure C.5: Details of the network interface.

The cut-through and loopback router simply recognizes addresses destined for the local processor node or memory, and passes that data directly to its destination without going through the XPORT or PHY layers. Data through the cut-through and loopback paths is always guaranteed to be single-word length. If the cut-through or loopback destination is unable to accept data, it must send a stall signal to the sender, and there must be sufficient buffering in the cut-through and loopback router to compensate for the time-of-flight of the stall signal. All data not destined for the cut-through or loopback paths is sent on to an outgoing packet buffer.

The outgoing packet buffer is responsible for regulating the flow of data from the processor node and migration manager going into the XPORT layer. It must be fairly large: big enough to hold a few maximum length packets. It also must be fairly flexible, since most packets are either going to be a couple words in size, or a couple hundred words in size. Finally, the packet buffer must implement the following contract with the migration manager and the processor node: once data of a given length has been accepted for transfer into the buffer, it must be able to accept all of that data. If it cannot, it must refuse any data from the sender, and it is the sender's responsibility to retry sending the data. In the case of the processor node, the stall will cause the sending thread to retire to the scheduler list and a retry occurs when the scheduler tries running the stalled thread again. In the case of the migration manager, the scheduler co-processor is responsible for implementing some kind of back-off and retry scheme in software. The packet buffer may also optionally implement packet merging for data going to the same destination. It must always preserve the temporal order of data after a merge, and it cannot merge atomic `EXCH` operations.

Data accepted into the packet buffer is then forwarded to the XPORT layer, but only when there is space available in the XPORT buffer. The

XPORT layer adds the requisite checksum, ECC and PHY-layer routing headers before storing the data inside the transport buffer. A processor-node unique sequence number is also added to the message while being stored into the transport buffer in order to implement reliable, idempotent, and in-order delivery.

The protocol used for reliable idempotent delivery relies on unique sequence numbers and acknowledge/forget tokens. It is the protocol developed by Jeremy Brown and J.P. Grossman of the MIT AI Lab with an additional level of numbering to guarantee the in-order delivery of messages. [GB02] Figure C.6 illustrates the simplified protocol without in-order delivery. Every message in the sender is assigned a unique sequence number. This number may be guaranteed to be unique by simply using a very large (64-bit) counter and incrementing it once for each data packet. The sender remembers the data packet and the sequence number, even after the packet is sent. Once the receiver gets the packet, it remembers the sequence number and the sender's context ID, and passes the data portion on to the destination within the node. The receiver then returns an ACK packet to the sender as soon as possible with the sequence number as the payload; when the sender sees the ACK packet, it knows it can safely forget about the buffered packet; reliable delivery has occurred. If, however, after some timeout period an ACK is not received, the sender must resend its data packet. If it is the case that the ACK was not received because the ACK path was blocked or corrupted, then a risk of a double-write of data occurs. This is averted, though, because the receiver keeps track of the the sender's sequence number; any incoming packet with a non-unique sequence number from a particular processor node is discarded, and another ACK generated. The FORGET packet is required by the receiver so it knows when it can retire sequence numbers and stop sending ACK packets. Thus, whenever a sender receives an ACK packet, it immediately turns

around a FORGET packet to the source of the ACK packet with the sequence number of the ACK packet as the payload.

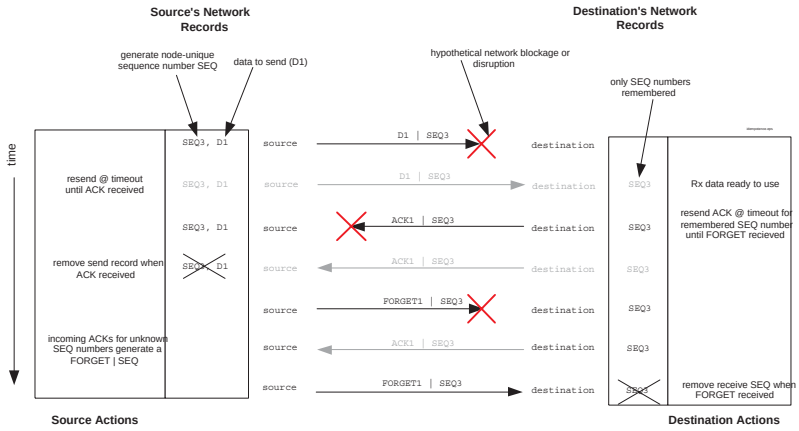


Figure C.6: Idempotence and reliable data delivery protocol in detail for a single transaction. Lines in gray are “retry” lines that would not happen in an ideal setting.

The basic protocol outlined above does not guarantee the in-order delivery of packets to a destination. Packets can be re-ordered by the fact that any packet in a sequence of packets could fail to be delivered on the first try. My tweak on the protocol is to include an additional queue ordering number in each packet. The queue ordering number starts at zero and is incremented each time a packet is sent for a given sequence number. The receiver’s job is to recreate the original ordering of the packets using the queue ordering numbers. An additional message, FORGET_CONNECTION, is required to signal when a sequence number can be forgotten. Sequence numbers can only be forgotten after sufficient time has passed to guarantee that the last packet sent in the protocol has either succeeded or failed. This is easy to determine because the network is circuit switched and delivery times are inherently bounded. A packet’s delivery is assumed to have failed if no acknowledgment

is received after a period of time has passed equal to the round trip time plus acknowledgment overhead time.

The network interface is structured to have one dedicated data packet transmit port, one dedicated ACK and FORGET packet transmit port, and two receive ports. This structure helps regulate the flow of data onto the network, and ensure that ACK and FORGET packets (which are smaller than data packets and thus able to be sent at a higher rate) have a greater chance of getting into the network. This structure also helps alleviate port contention at the receivers by limiting the peak rate of message injection to be strictly lower than the peak rate of message acceptance. Refer to section C.3 for more details on how the routing headers are structured and the number and types of wires used to interface to the network.

The format of the network packets is documented in figure C.7. The most important information to glean from this diagram is that ACK and FORGET packets are the shortest packets, and that an abbreviated "short data packet" is available for the very common special case that the packet contains a payload of one word. The short data packet is 20% shorter than a long data packet of length one because it combines the ECC field and the sequence number into the same word, and removes the length field entirely. The ECC/checksum field can be shorter for these packets because there is less data over which to checksum and correct.

C.3 Network Topology and Implementation

The network topology and its specific implementation parameters are free to vary depending upon the user's end requirements. The XPORT protocol discussed in section C.2 assumes that the network is circuit switched, *i.e.* it has no buffering or reordering, and that it is quite unreliable; contention for

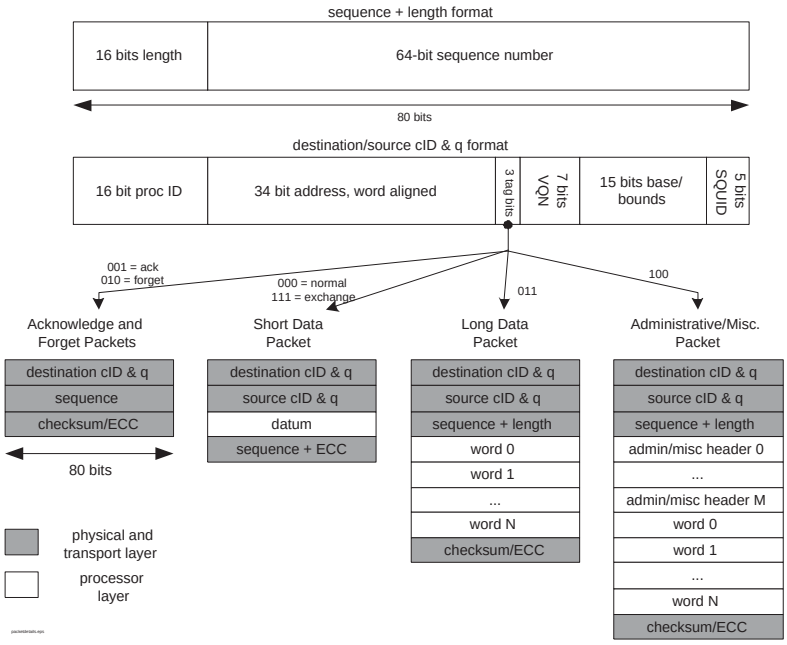


Figure C.7: Details of packet formats. Note that in the destination/source cID and queue headers, it is very important that the processor ID be in the MSB and co-located with the address field, since implementations may push bits between the address and PID fields to increase the number of routable processor nodes or to increase the amount of memory per node.

ports and routing resources is indistinguishable from hardware failures from the sender's standpoint. This unreliability is not as bad as it sounds; under light loading conditions, connections are rapidly and reliably established, and connection performance degrades gradually as congestion increases. With a properly designed transport protocol, however, this phenomenon can be used as feedback to throttle the message insertion rate. The network is also quite robust in the face of hardware failures, since it is designed from the ground-up to cope with such scenarios. The ADAM System Simulator implements a network topology based upon the METRO network, described in great detail by [DeH93] and by [WC01].

The principal advantage of circuit-switched networks over packet-routed networks is latency performance and simplicity of implementation. No buffers are required at the routers to handle port contention: the message is simply dropped, and the sender is responsible for re-sending the message. This is because the network does not have to guarantee message delivery. Also, with the correct choice of network topology, routing can happen at wave-propagation speeds, such that the wire delay, even over short runs, is the dominant latency component of the network.

One of the disadvantages of circuit-switched networks is that the network is very bandwidth-inefficient if the minimum time required to establish a connection is long compared to the time required to deliver the message data. This kind of scenario may happen in a room-sized computer where the velocity of electromagnetic waves in a copper waveguide forces the minimum time to establish a connection to be several tens or hundreds of processor clock periods long. This is particularly painful in the case that a connection is blocked by router contention near the destination, because routing resources were consumed and locked-down throughout the body of the network for the length of the connection. The worst-case scenario occurs when several

senders are trying to communicate with a single remote hot-spot on a deep circuit-switched network, causing multiple messages to route easily through the body of the network, consuming resources, only to get dropped near the destination. André DeHon of Caltech suggested that a hybrid buffered packet and circuit-switched network may be a good solution under these conditions: one may insert packet buffer stations that store a limited number of packets (and may freely drop excess packets) at intermediate "checkpoints" along the network. In the hot-spot scenario, messages route easily through most of the network and are stored at the checkpoint nearest the hot-spot, and only the segment of network between the hot-spot and the nearest checkpoint suffers degenerative congestion. The distance between checkpoints and the depth of the checkpoint buffers are parameters that depend heavily upon the implementation technology, and in particular, the ratio of the information propagation time to the temporal length of the average message.

Another technique for reducing the cost of circuit-switching, suggested by J.P. Grossman of the MIT AI Lab, is to use worm-hole routing. One can build a worm-hole routed network using the same fast router structures and protocols as a circuit switched network, but instead of holding the circuit up until the receiver tears it down with an acknowledge, the connection is torn up as the tail of the message is routed. This method of routing requires that separate ACK and FORGET routes have to be re-established, but this price is relatively small. The reliable delivery and idempotence protocol outlined in section C.2 can handle blocked ACK and FORGET packets. Also, the data payload is delivered on the first packet to arrive at the destination, so even if the ACK and FORGET packets take a while to work their way through the system, the effective delivery latency is still just the price of a one-way trip.

The recommended network topology for ADAM implementations is a hybrid topology similar to that suggested in [DeH90]. The topology of an

on-chip or local network should be a radix-4 dilation-2 bidelta network as described by [DeH93]. For off-chip or longer-distance networks in larger systems, some bandwidth thinning is required for scalability. The basic router components designed for the on-chip networks can be re-used to implement a more scalable fat-tree topology as described in [DeH90] and in [WC01]. The parameters assumed by the year 2010 implementation described in section C.2 implies that the off-chip network could run at speeds of 2-4 GHz, double-edge clocked. Assuming that 92 bits are required to represent one flit, 46 differential pairs could transmit a full 80-bit word plus ECC, sync and clock every processor cycle. There will probably be enough pins on a package in the year 2010 to implement dozens of these high speed links per chip.

Finally, it is recommended that the implementation use a single frequency reference and a mesochronous (phase-insensitive) timing scheme for the entire machine. This single reference may have redundancy built into it, or auxiliary resonators distributed throughout the machine, to prevent a meltdown or power grid failure due to inductive kickback as a result of clock failure. The implementation would require an initial self-calibration phase where receivers determine the optimal sampling phase, and perhaps even require periodic (on the order of minutes or hours) re-calibrations where the machine is paused for a microsecond or two to compensate for material property changes over time and temperature. The principle advantage of using a mesochronous single frequency source scheme is that one can remove the metastability resolution time from the network timing budget, and the secondary advantage is that it simplifies the implementation of the physical layer, as plesiochronous implementations require some complexity to handle the case when the integrated frequency error causes a cycle to be lost between nodes. The impact of metastability resolution and synchronization on the latency of a router node

should not be underestimated, especially if the router is operating at near wave-propagation speeds. In the SGI SPIDER router chip used by the Origin 2000 supercomputers [Gal96], 17.5 ns out of a total 40 ns pin to pin latency is burned in the synchronizer. The problem with metastability is that there is nothing one can do about it except wait for the values to settle, and the settling time is an inverse exponential to the magnitude of the difference between the initial voltage and the fixed-point metastable voltage. A mesochronous system side-steps this issue by calibrating the sample time at clock boundaries to give the biggest margin versus the metastable voltage.

Appendix D

Opcodes

> Implementing someone else's specification is the moral equivalent of translating fifty VCR user's manuals from English to Japanese.
>
> —*bunnie*

D.1 General Notes

RTL descriptions of opcode operations are given in blocking form; i.e., the following lines of code

```
PC ← PC + 1
qc ← PC
PC ← PC + offset
```

stores the value of the initial `PC + 1` into `qc`, and the value of the initial `PC + 1 + offset` into `PC`.

Also, note that if no PC operation is specified, a default operation of PC ← PC + 1 is implied, and that an exception can be thrown as a result of the PC increment if the PC enters into a protected or invalid code region.

D.2 Lazy Instructions

The following instructions may require multiple cycles to complete execution *and* do not stall the program counter (some instructions will require multiple cycles, but stall the PC until they are complete). The most important thing to note is that these instructions in fact do not guarantee how long it will take to complete. Two instructions started in an overlapping manner may complete out of order. For example, the code

```
SPAWNC qn, label1, q0
SPAWNC qn, label2, q0
```

May result with the capability for the label1 thread returned after the capability for the label2 thread. If the order of the return values matters, it is recommended that a blocking intermediate queue move operation be employed:

```
SPAWNC qn, label1, q0
MOVE q0, q1
SPAWNC qn, label2, q0
MOVE q0, q1
```

Execution will block each time on the MOVE q0, q1 instruction until q0 has a value.

This behavior of a multicycle instruction is referred to as “lazy”. The following instructions are lazy:

```
SPAWN
SPAWNC
ALLOCATE
ALLOCATEC
```

D.3 Instruction Summary

Integer Arithmetic Instructions:

```
ADD qa, qb, qc
SUB qa, qb, qc
MUL qa, qb, qc
DIV qa, qb, qc
ADDC qa, n, qc
SUBC qa, n, qc
MULC qa, n, qc
DIVC qa, n, qc
```

Logical Operator Instructions:

```
AND qa, qb, qc
OR qa, qb, qc
XOR qa, qb, qc
NOT qa, qc
ANDC qa, n, qc
ORC qa, n, qc
XORC qa, n, qc
SHL qa, qb, qc
SHR qa, qb, qc
SRA qa, qb, qc
SHLC qa, n, qc
SHRC qa, n, qc
SRAC qa, n, qc
```

Integer Comparison Instructions:

```
SEQ qa, qb, qc
SNE qa, qb, qc
SLT qa, qb, qc
SGT qa, qb, qc
SLE qa, qb, qc
SGE qa, qb, qc
SIC qa, qc
SEQC qa, n, qc
SNEC qa, n, qc
SLTC qa, n, qc
SGTC qa, n, qc
SLEC qa, n, qc
SGEC qa, n, qc
```

Floating point to Integer Conversions:

```
TOINT qa, qc
```

```
   TOREAL qa, qc
```
Floating Point Arithmetic Instructions:
```
   FADD qa, qb, qc
   FSUB qa, qb, qc
   FMUL qa, qb, qc
   FDIV qa, qb, qc
   FADDC qa, n, qc
   FSUBC qa, n, qc
   FMULC qa, n, qc
   FDIVC qa, n, qc
```
Floating Point Comparison Instructions:
```
   FSEQ qa, qb, qc
   FSNE qa, qb, qc
   FSLT qa, qb, qc
   FSGT qa, qb, qc
   FSLE qa, qb, qc
   FSGE qa, qb, qc
   FSEQC qa, n, qc
   FSNEC qa, n, qc
   FSLTC qa, n, qc
   FSGTC qa, n, qc
   FSLEC qa, n, qc
   FSGEC qa, n, qc
```
Branch and Jump Instructions:
```
   BR label
   BRL label, qc
   BRZ qa, label
   BRNZ qa, label
   BRNE qa, label
   BREL qa
   JMP qa
```
Internal Data Manipulation Instructions:
```
   MOVE qa, qc
   MOVECF n, qc
   MOVECL n, qc
   MOVECI n, qc
   MOVECS n, qc
   MOVECC n, qc
   PACKN qa, qb, qc, n
   PACKH qa, qb, qc
   PACKL qa, qb, qc
   PACKI qa, qb, qc
```

```
   UNPACK qa, qb, qc
   UNPACKC qa, n, qc
   EXTAG qa, qc
   SETTAG qa, qb, qc
```
Queue Management Instructions:
```
   FLUSHQ qc
   SPAWN qa, qb, qc
   SPAWNC qa, label, qc
   SPAWNL qa, qb, qc
   MAPQ qa, qb, qc
   MAPQC qa, qb, qc
   MAPSQ qa, qb
   MAPDROP n
   UNMAPQ n
   CONSUME qa
   SEMPTY qa, qc
   EEQ qc
```
Thread and Context Management Instructions:
```
   PROCID qc
   LDCODE qa, qc
   OSIZE n
```
Memory Instructions:
```
   PTRSIZE qa, qc
   ALLOCATE qa, qb, qc
   ALLOCATEC qa, n, qc
   MML qa, qb
   MMS qa, qb
   EXCH qa, qb, qc
   PARCEL qa, qb, qc
   MSYNC
```
Mode and Exception Handling Instructions:
```
   GETSTAT qc
   SETSTAT qa
   GETEX qc
   SETEX qa
   THROW
```
Miscellaneous Instructions:
```
   RANDOM qc
   HINT t,hint
```

ADD

ADD qa, qb, qc

Description:

ADD (addition) takes the sum of qa and qb and returns the result in qc. qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. Also, qa may be a capability and qb may be a word, in which case the result will be a capability. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised. If qa is a capability and the add operation with word in qb is not permitted or results in an invalid capability, an operation exception is raised and the result in qc is an invalid capability.

If qa is non-copyable capability, then a successful ADD operation dequeues qa even if the copy/clobber modifier for qa is set to copy and an exception is thrown.

The ADD operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
    qc ← qa + qb
elif( type(qa,qb) == packed int )
    qc.a ← qa.a + qb.a
    qc.b ← qa.b + qb.b
elif( type(qa,qb) == (packed char or packed short) )
    qc.a ← qa.a + qb.a
    qc.b ← qa.b + qb.b
    qc.c ← qa.c + qb.c
    qc.d ← qa.d + qb.d
elif( (type(qa) == capability) && (type(qb) == word) )
    temp ← qa + SEXT(qb & ADDRMASK)
    if( temp is valid )
       qc ← temp
       if( qa == non-copyable )
          forceDequeue( qa ) // flag error if copy bit is set on qa
    else
```

```
        throw operation exception
        qc ← invalid
else
    throw type exception
```

Exceptions:

Type exception, operation exception, and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in `qc`'s type field.

ADDC qa, n, qc

Description:

ADDC (addition with constant) takes the sum of qa and n and returns the result in qc. qa can be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. In the case of packed types, the same constant is added to each sub-integer. Also, qa may be a capability, in which case the result will be a capability. If qa is a capability and the add operation with word in qb is not permitted or results in an invalid capability, an operation exception is raised and the result in qc is an invalid capability.

If qa is non-copyable capability, then a successful ADDC operation dequeues qa even if the copy/clobber modifier for qa is set to copy and an exception is thrown.

The ADDC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    qc ← qa + SEXT(n)
elif( type(qa) == packed int )
    qc.a ← qa.a + n
    qc.b ← qa.b + n
elif( type(qa) == (packed char or packed short) )
    qc.a ← qa.a + n
    qc.b ← qa.b + n
    qc.c ← qa.c + n
    qc.d ← qa.d + n
elif( type(qa) == capability )
    temp ← qa + SEXT(n & ADDRMASK)
    if( temp is valid )
       qc ← temp
       if( qa == non-copyable )
          forceDequeue( qa ) // flag error if copy bit is set on qa
    else
       throw operation exception
       qc ← invalid
else
    throw type exception
```

Exceptions:

Type exception, operation exception, and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in `qc`'s type field.

SUB

SUB qa, qb, qc

Description:

SUB (subtraction) takes the difference of qa and qb and returns the result in qc. qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. Also, qa may be a capability and qb may be a word, in which case the result will be a capability. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised. If qa is a capability and the add operation with word in qb is not permitted or results in an invalid capability, an operation exception is raised and the result in qc is an invalid capability.

If qa is non-copyable capability, then a successful SUB operation dequeues qa even if the copy/clobber modifier for qa is set to copy, and an exception is thrown.

The SUB operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
    qc ← qa - qb
elif( type(qa,qb) == packed int )
    qc.a ← qa.a - qb.a
    qc.b ← qa.b - qb.b
elif( type(qa,qb) == (packed char or packed short) )
    qc.a ← qa.a - qb.a
    qc.b ← qa.b - qb.b
    qc.c ← qa.c - qb.c
    qc.d ← qa.d - qb.d
elif( (type(qa) == capability) && (type(qb) == word) )
    temp ← qa - SEXT(qb & ADDRMASK)
    if( temp is valid )
       qc ← temp
       if( qa == non-copyable )
          forceDequeue( qa )
    else
```

```
        throw operation exception
        qc ← invalid
else
    throw type exception
```

Exceptions:

Type exception, operation exception, and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in `qc`'s type field.

SUBC qa, n, qc

Description:

SUBC (subtraction with constant) takes the difference of qa and n and returns the result in qc. qa can be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. In the case of packed types, the same constant is subtracted from each sub-integer. Also, qa may be a capability, in which case the result will be a capability. If qa is a capability and the add operation with word in qb is not permitted or results in an invalid capability, an operation exception is raised and the result in qc is an invalid capability.

If qa is non-copyable capability, then a successful SUBC operation dequeues qa even if the copy/clobber modifier for qa is set to copy and an exception is thrown.

The SUBC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    qc ← qa - SEXT(n)
elif( type(qa) == packed int )
    qc.a ← qa.a - n
    qc.b ← qa.b - n
elif( type(qa) == (packed char or packed short) )
    qc.a ← qa.a - n
    qc.b ← qa.b - n
    qc.c ← qa.c - n
    qc.d ← qa.d - n
elif( type(qa) == capability )
    temp ← qa - SEXT(n & ADDRMASK)
    if( temp is valid )
       qc ← temp
       if( qa == non-copyable )
          forceDequeue( qa ) // flag error if copy bit is set on qa
    else
       throw operation exception
       qc ← invalid
else
    throw type exception
```

Exceptions:

Type exception, operation exception, and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in `qc`'s type field.

MUL

MUL qa, qb, qc

Description:

MUL (multiplication) takes the product of qa and qb and returns the lowest bits of the result in qc. qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised.

The MUL operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
     qc ← (qa * qb) & 0xFFFFFFFFFFFFFFFF
elif( type(qa,qb) == packed int )
     qc.a ← (qa.a * qb.a) & 0xFFFFFFFF
     qc.b ← (qa.b * qb.b) & 0xFFFFFFFF
elif( type(qa,qb) == (packed char or packed short) )
     qc.a ← (qa.a * qb.a) & 0xFFFF
     qc.b ← (qa.b * qb.b) & 0xFFFF
     qc.c ← (qa.c * qb.c) & 0xFFFF
     qc.d ← (qa.d * qb.d) & 0xFFFF
else
     throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

MULC qa, n, qc

Description:

MULC (multiplication with constant) takes the product of qa and n and returns the lowest bits of the result in qc. qa can be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. In the case of packed types, the same constant is multiplied to each sub-integer.

The MULC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    qc ← (qa * n) & 0xFFFFFFFFFFFFFFFF
elif( type(qa) == packed int )
    qc.a ← (qa.a * n) & 0xFFFFFFFF
    qc.b ← (qa.b * n) & 0xFFFFFFFF
elif( type(qa) == (packed char or packed short) )
    qc.a ← (qa.a * n) & 0xFFFF
    qc.b ← (qa.b * n) & 0xFFFF
    qc.c ← (qa.c * n) & 0xFFFF
    qc.d ← (qa.d * n) & 0xFFFF
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

DIV qa, qb, qc

Description:

DIV (integer divide) takes the division of qa and qb and returns the result in qc. Non-integer results are truncated. qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised. If the divisor qb is zero, a divide by zero exception is thrown and qc is marked as invalid, with the specific packed component of qc that is erroneous marked as overflowed.

The DIV operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
     if(qb == 0)
        throw divide-by-zero exception
        type(qc) ← invalid, overflow.a
     else
        qc ← qa / qb
elif( type(qa,qb) == packed int )
     if(qb.a == 0)
        throw divide-by-zero exception
        type(qc.a) ← invalid, overflow.a
     else
        qc.a ← qa.a / qb.a
     if(qb.b == 0)
        throw divide-by-zero exception
        type(qc.b) ← invalid, overflow.b
     else
        qc.b ← qa.b / qb.b
elif( type(qa,qb) == (packed char or packed short) )
     if(qb.a == 0)
        throw divide-by-zero exception
        type(qc.a) ← invalid, overflow.a
     else
        qc.a ← qa.a / qb.a
     if(qb.b == 0)
```

```
        throw divide-by-zero exception
        type(qc.b) ← invalid, overflow.b
    else
        qc.b ← qa.b / qb.b
    if(qb.c == 0)
        throw divide-by-zero exception
        type(qc.c) ← invalid, overflow.c
    else
        qc.c ← qa.c / qb.c
    if(qb.d == 0)
        throw divide-by-zero exception
        type(qc.d) ← invalid, overflow.d
    else
        qc.d ← qa.d / qb.d
else
    throw type exception
```

Exceptions:

Type exception, and divide-by-zero exception.

Qualifiers:

None.

Notes:

None.

DIVC qa, n, qc

Description:

DIVC (division with constant) takes the division of qa and n and returns the result in qc. qa can be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. In the case of packed types, the same constant is multiplied to each sub-integer. If the divisor n is zero, a divide by zero exception is thrown and qc is marked as invalid, with the specific packed component of qc that is erroneous marked as overflowed.

The DIVC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    if(n == 0)
        throw divide-by-zero exception
        type(qc) ← invalid, overflow.a
    else
        qc ← qa / SEXT(n)
elif( type(qa) == packed int )
    if(n == 0)
        throw divide-by-zero exception
        type(qc.a) ← invalid, overflow.a
        type(qc.b) ← invalid, overflow.b
    else
        qc.a ← qa.a / n
        qc.b ← qa.b / n
elif( type(qa) == (packed char or packed short) )
    if(n == 0)
        throw divide-by-zero exception
        type(qc.a) ← invalid, overflow.a
        type(qc.b) ← invalid, overflow.b
        type(qc.c) ← invalid, overflow.c
        type(qc.d) ← invalid, overflow.d
    else
        qc.a ← qa.a / n
        qc.b ← qa.b / n
        qc.c ← qa.c / n
```

```
        qc.d ← qa.d / n
else
    throw type exception
```

Exceptions:

Type exception, divide-by-zero exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in `qc`'s type field.

AND,OR,XOR

AND,OR,XOR qa, qb, qc

Description:

`AND`, `OR`, and `XOR` perform bitwise operations on `qa` and `qb` and returns the result in `qc`. `qa` and `qb` must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in `qc` will have the same type as its predecessors. If `qa` or `qb` have incompatible types, `qc` will be tagged as invalid and a type exception raised.

The `AND,OR,XOR` operation is only executed if both `qa` and `qb` operands are available and there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
OP is one of bitwise AND, OR, XOR
if( type(qa,qb) == word )
     qc ← qa OP qb
elif( type(qa,qb) == packed int )
     qc.a ← qa.a OP qb.a
     qc.b ← qa.b OP qb.b
elif( type(qa,qb) == (packed char or packed short) )
     qc.a ← qa.a OP qb.a
     qc.b ← qa.b OP qb.b
     qc.c ← qa.c OP qb.c
     qc.d ← qa.d OP qb.d
else
     throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

NOT qa, qc

Description:

NOT performs a bitwise inversion on qa and returns the result in qc. qa must be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The NOT operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
     qc ← ~qa
elif( type(qa) == packed int )
     qc.a ← ~qa.a
     qc.b ← ~qa.b
elif( type(qa,qb) == (packed char or packed short) )
     qc.a ← ~a.a
     qc.b ← ~qa.b
     qc.c ← ~qa.c
     qc.d ← ~qa.d
else
     throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

ANDC,ORC,XORC

ANDC,ORC,XORC qa, n, qc

Description:

ANDC, ORC, and XORC perform a bitwise operation on qa and a sign-extended n and returns the result in qc. qa can be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessors. In the case of packed types, the same constant is operated on each sub-integer.

The ANDC, ORC, XORC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
OP is one of bitwise AND, OR, XOR
if( type(qa) == word )
     qc ← qa OP SEXT(n)
elif( type(qa) == packed int )
     qc.a ← qa.a OP n
     qc.b ← qa.b OP n
elif( type(qa) == (packed char or packed short) )
     qc.a ← qa.a OP n
     qc.b ← qa.b OP n
     qc.c ← qa.c OP n
     qc.d ← qa.d OP n
else
     throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SHL qa, qb, qc

Description:

SHL (shift-left) performs a logical left-shift on the contents of qa by the number of digits specified in qb, and returns the result in qc. Bits shifted off the left are thrown away, and zeroes are shifted in from the right. qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised.

The SHL operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
     qc ← qa ≪ (qb & 0x3F)
elif( type(qa,qb) == packed int )
     qc.a ← qa.a ≪ (qb.a & 0x1F)
     qc.b ← qa.b ≪ (qb.b & 0x1F)
elif( type(qa,qb) == (packed char or packed short) )
     qc.a ← qa.a ≪ (qb.a & 0xF)
     qc.b ← qa.b ≪ (qb.b & 0xF)
     qc.c ← qa.c ≪ (qb.c & 0xF)
     qc.d ← qa.d ≪ (qb.d & 0xF)
else
     throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SHLC qa, n, qc

Description:

SHLC (shift left by constant) performs a logical left-shift on the contents of qa by the number of digits specified in n, and returns the result in qc. Bits shifted off the left are thrown away, and zeroes are shifted in from the right. qa must be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. In the case that qa is a packed type, each subword will be shifted left by the same amount. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The SHLC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
     qc ← qa ≪ (n & 0x3F)
elif( type(qa) == packed int )
     qc.a ← qa.a ≪ (n & 0x1F)
     qc.b ← qa.b ≪ (n & 0x1F)
elif( type(qa) == (packed char or packed short) )
     qc.a ← qa.a ≪ (n & 0xF)
     qc.b ← qa.b ≪ (n & 0xF)
     qc.c ← qa.c ≪ (n & 0xF)
     qc.d ← qa.d ≪ (n & 0xF)
else
     throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SHR qa, qb, qc

Description:

SHR (logical shift right)performs a logical right-shift on the contents of qa by the number of digits specified in qb, and returns the result in qc. Bits shifted off the right are thrown away, and zeroes are shifted in from the left. qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised.

The SHR operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
     qc ← qa ≫ (qb & 0x3F)
elif( type(qa,qb) == packed int )
     qc.a ← qa.a ≫ (qb.a & 0x1F)
     qc.b ← qa.b ≫ (qb.b & 0x1F)
elif( type(qa,qb) == (packed char or packed short) )
     qc.a ← qa.a ≫ (qb.a & 0xF)
     qc.b ← qa.b ≫ (qb.b & 0xF)
     qc.c ← qa.c ≫ (qb.c & 0xF)
     qc.d ← qa.d ≫ (qb.d & 0xF)
else
     throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SHRC qa, n, qc

Description:

SHRC (logical shift right by constant) performs a logical right-shift on the contents of qa by the number of digits specified in n, and returns the result in qc. Bits shifted off the right are thrown away, and zeroes are shifted in from the left. qa must be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. In the case that qa is a packed type, each subword will be shifted left by the same amount. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The SHRC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    qc ← qa ≫ (n & 0x3F)
elif( type(qa) == packed int )
    qc.a ← qa.a ≫ (n & 0x1F)
    qc.b ← qa.b ≫ (n & 0x1F)
elif( type(qa) == (packed char or packed short) )
    qc.a ← qa.a ≫ (n & 0xF)
    qc.b ← qa.b ≫ (n & 0xF)
    qc.c ← qa.c ≫ (n & 0xF)
    qc.d ← qa.d ≫ (n & 0xF)
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SRA qa, qb, qc

Description:

SRA (arithmetic shift right) performs an arithmetic (sign-preserving) right-shift on the contents of qa by the number of digits specified in qb, and returns the result in qc. Bits shifted off the right are thrown away, and the value of the sign bit is shifted in from the left (zero if the number being shifted is positive, one if the number being shifted is negative). qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised.

The SRA operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
     qc ← qa SRA (qb & 0x3F)
elif( type(qa,qb) == packed int )
     qc.a ← qa.a SRA (qb.a & 0x1F)
     qc.b ← qa.b SRA (qb.b & 0x1F)
elif( type(qa,qb) == (packed char or packed short) )
     qc.a ← qa.a SRA (qb.a & 0xF)
     qc.b ← qa.b SRA (qb.b & 0xF)
     qc.c ← qa.c SRA (qb.c & 0xF)
     qc.d ← qa.d SRA (qb.d & 0xF)
else
     throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SRAC qa, n, qc

Description:

SRAC (arithmetic shift right by constant) performs an arithmetic right-shift on the contents of qa by the number of digits specified in n, and returns the result in qc. Bits shifted off the right are thrown away, and the value of the sign bit is shifted in from the left (zero if the number being shifted is positive, one if the number being shifted is negative). qa must be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. In the case that qa is a packed type, each subword will be shifted left by the same amount. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The SRAC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    qc ← qa SRA (n & 0x3F)
elif( type(qa) == packed int )
    qc.a ← qa.a SRA (n & 0x1F)
    qc.b ← qa.b SRA (n & 0x1F)
elif( type(qa) == (packed char or packed short) )
    qc.a ← qa.a SRA (n & 0xF)
    qc.b ← qa.b SRA (n & 0xF)
    qc.c ← qa.c SRA (n & 0xF)
    qc.d ← qa.d SRA (n & 0xF)
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SEQ,SLT,SLE

SEQ,SLT,SLE qa, qb, qc

Description:

SEQ, SLT, and SLE perform magnitude comparisons on its arguments and produce a binary result. SEQ test if qa and qb are equal; SLT tests if qa is less than qb; and SLE tests if qa is less than or equal to qb˙qa and qb must be of the same integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised.

The Sxx operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
OP is one of arithmetic =, <, ≤:
if( type(qa,qb) == word )
     qc ← qa OP qb ?  1 :  0
elif( type(qa,qb) == packed int )
     qc.a ← qa.a OP qb.a ?  1 :  0
     qc.b ← qa.b OP qb.b ?  1 :  0
elif( type(qa,qb) == (packed char or packed short) )
     qc.a ← qa.a OP qb.a ?  1 :  0
     qc.b ← qa.b OP qb.b ?  1 :  0
     qc.c ← qa.c OP qb.c ?  1 :  0
     qc.d ← qa.d OP qb.d ?  1 :  0
else
     throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

SIC qa, qc

Description:

SIC tests if qa is a capability. If it is, a word type 1 is put into qc. Otherwise, a word type 0 is put into qc.

The SIC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == capability )
    qc ← 1
else
    qc ← 0
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

SEQC,SLTC,SLEC

SEQC,SLTC,SLEC qa, qb, qc

Description:

SEQC, SLTC, and SLEC perform magnitude comparisons on its arguments and produce a binary result. SEQC test if qa and n are equal; SLTC tests if qa is less than n; and SLEC tests if qa is less than or equal to n. qa must be of an integer type (word, packed int, packed short, or packed char), in which case the result in qc will have the same type as its predecessor. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The SxxC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
OP is one of arithmetic =, <, ≤:
if( type(qa,qb) == word )
     qc ← qa OP n ?  1 :  0
elif( type(qa,qb) == packed int )
     qc.a ← qa.a OP n ?  1 :  0
     qc.b ← qa.b OP n ?  1 :  0
elif( type(qa,qb) == (packed char or packed short) )
     qc.a ← qa.a OP n ?  1 :  0
     qc.b ← qa.b OP n ?  1 :  0
     qc.c ← qa.c OP n ?  1 :  0
     qc.d ← qa.d OP n ?  1 :  0
else
     throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

TOINT	qa, qc

Description:

TOINT (floating point to integer convert) converts the floating-point value in qa to an integer stored in qc. Conversion is done using the truncation or "round to zero" method, so that the number 9.6 is converted to 9, and the number -2.8 is converted to -2. Overflow in either sign extreme results in qc having the maximum sized integer of the appropriate sign and the overflow bit being set in qc's type field. qa must be of the floating point type, and the result in qc is of type word. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The TOINT operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == floating-point )
    qc ← (word) qa
    type(qc) ← word
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field. Attempting to convert $+\infty$ will result in the largest positive representable integer in qc and set the overflow bit of qc. Likewise, converting $-\infty$ will result in the most negative representable integer in qc and set the overflow bit of qc.

Attempting to convert NaN's will result in qc having an invalid type.

TOREAL

TOREAL qa, qc

Description:

TOREAL (integer to floating point convert) converts the integer value in qa to the nearest representable floating-point value stored in qc. qa must be of the word type, and the result in qc is of the floating point type. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The TOREAL operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    qc ← (floating-point) qa
    type(qc) ← floating-point
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

FADD

FADD qa, qb, qc

Description:

FADD (floating-point addition) takes the sum of qa and qb and returns the result in qc. qa and qb must be of the floating-point type, and the result qc is of the floating point type.

The FADD operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == floating-point )
    qc ← qa + qb
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If any operand is a NaN, the result will be NaN.

FADDC

FADDC qa, n, qc

Description:

FADDC (floating-point addition with constant) takes the sum of qa and n and returns the result in qc. qa must be of the floating-point type, and the result qc is of the floating point type.

The FADDC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == floating-point )
    qc ← qa + n
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If qa is a NaN, the result will be NaN.

FSUB

FSUB qa, qb, qc

Description:

FSUB (floating-point subtraction) takes the difference of qa and qb and returns the result in qc. qa and qb must be of the floating-point type, and the result qc is of the floating point type.

The FSUB operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == floating-point )
    qc ← qa - qb
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If any operand is a NaN, the result will be NaN.

FSUBC qa, n, qc

Description:

FSUBC (floating-point addition with constant) takes the difference of qa and n and returns the result in qc. qa must be of the floating-point type, and the result qc is of the floating point type.

The FSUBC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == floating-point )
    qc ← qa - n
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If qa is a NaN, the result will be NaN.

FMUL

FMUL qa, qb, qc

Description:

`FMUL` (floating-point multiply) takes the product of `qa` and `qb` and returns the result in `qc`. `qa` and `qb` must be of the floating-point type, and the result `qc` is of the floating point type.

The `FMUL` operation is only executed if both `qa` and `qb` operands are available and there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == floating-point )
    qc ← qa * qb
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in `qc`'s type field.

If any operand is a NaN, the result will be NaN.

FMULC qa, n, qc

Description:

FMULC (floating-point multiply with constant) takes the product of qa and n and returns the result in qc. qa must be of the floating-point type, and the result qc is of the floating point type.

The FMULC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == floating-point )
    qc ← qa + n
else
    throw type exception
```

Exceptions:

Type exception and overflow exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If qa is a NaN, the result will be NaN.

FDIV qa, qb, qc

Description:

FDIV (floating-point division) divides qa by qb and returns the quotient in qc. qa and qb must be of the floating-point type, and the result qc is of the floating point type. If qb is zero, a divide-by-zero exception is thrown and the result qc is tagged as invalid.

The FDIV operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == floating-point )
    qc ← qa / qb
else
    throw type exception
```

Exceptions:

Type exception, overflow exception, and divide-by-zero exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If any operand is a NaN, the result will be NaN.

FDIVC

FDIVC qa, n, qc

Description:

FDIVC (floating-point divide by constant) divides qa by n and returns the result in qc. qa must be of the floating-point type, and the result qc is of the floating point type. If n is zero, a divide-by-zero exception is thrown and the result qc is tagged as invalid.

The FDIVC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == floating-point )
    qc ← qa / n
else
    throw type exception
```

Exceptions:

Type exception, overflow exception, and divide-by-zero exception.

Qualifiers:

None.

Notes:

Overflowed results also set the respective overflow bit in qc's type field.

If qa is a NaN, the result will be NaN.

FSEQ,FSLT,FSLE

FSEQ,FSLT,FSLE qa, qb, qc

Description:

FSEQ, FSLT, and FSLE perform magnitude comparisons on its arguments and produce a binary integer result. FSEQ test if qa and qb are equal; FSLT tests if qa is less than qb; and FSLE tests if qa is less than or equal to qb˙qa and qb must be of the floating-point type. The result qc is of type word. If qa or qb have incompatible types, qc will be tagged as invalid and a type exception raised.

The FSxx operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
OP is one of arithmetic =, <, ≤:
if( type(qa,qb) == floating-point )
     qc ← qa OP qb ?  1 :  0
     type(qc) ← word
else
     throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

If any of the operands are NaNs, the result is tagged as invalid.

FSEQC,FSLTC,FSLEC

FSEQC,FSLTC,FSLEC qa, qb, qc

Description:

FSEQC, FSLTC, and FSLEC perform magnitude comparisons on its arguments and produce a binary result. FSEQC test if qa and n are equal; FSLTC tests if qa is less than n; and FSLEC tests if qa is less than or equal to n. qa must be of the floating-point type, and the result in qc is of type word. If qa has an incompatible type, qc will be tagged as invalid and a type exception raised.

The FSxxC operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
OP is one of arithmetic =, <, ≤:
if( type(qa,qb) == floating-point )
    qc ← qa OP n ?  1 :  0
    type(qc) ← word
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

If qa is a NaN, the result is tagged as invalid.

BR offset

Description:

BR (unconditional branch) adds the number specified in the offset field to the incremented program counter. Execution immediately begins at the new PC value; there are no branch delay slots.

Operation:

```
PC ← PC + 1
PC ← PC + offset
```

Exceptions:

If the destination of the PC is in a protected or invalid page, an exception is thrown.

Qualifiers:

None.

Notes:

None.

BRL offset, qc

Description:

BRL (unconditional branch with link) adds the number specified in the offset field to the incremented program counter. Execution immediately begins at the new PC value; there are no branch delay slots. The incremented program counter offset relative to the start of code (be it method, object, or absolute-referenced) is stored in qc as a word data type; execution stalls if qc is full and applying backpressure.

The BRL operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
PC ← PC + 1
qc ← PC
PC ← PC + offset
```

Exceptions:

If the destination of the PC is in a protected or invalid page, an exception is thrown.

Qualifiers:

None.

Notes:

None.

BRZ qa, offset, hint

Description:

BRZ (branch if zero) adds the number specified in the offset field to the incremented program counter if the value in qa is zero; otherwise, the program counter is just incremented to the next instruction. qa must be of the word type. Execution immediately begins at the new PC value; there are no branch delay slots.

The BRZ operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    if( qa == 0 )
        PC ← PC + 1 + offset
    else
        PC ← PC + 1
```

Exceptions:

If the destination of the PC is in a protected or invalid page, an exception is thrown. A type exception is thrown if the type of qa is not word.

Qualifiers:

None.

Notes:

The hint field is an implementation-specific 8-bit number that serves as a branch prediction hint. The semantics of hint are such that an incorrect branch hint still leads to correct but slower execution. The actual value of hint is allowed to have cache-incoherent mutation during run-time as the dynamic hardware branch-predictor sees fit.

BRNZ

BRNZ qa, offset, hint

Description:

BRNZ (branch if not zero) adds the number specified in the offset field to the incremented program counter if the value in qa is not zero; otherwise, the program counter is just incremented to the next instruction. qa must be of the word type. Execution immediately begins at the new PC value; there are no branch delay slots.

The BRNZ operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    if( qa != 0 )
        PC ← PC + 1 + offset
    else
        PC ← PC + 1
```

Exceptions:

If the destination of the PC is in a protected or invalid page, an exception is thrown. A type exception is thrown if the type of qa is not word.

Qualifiers:

None.

Notes:

The hint field is an implementation-specific 8-bit number that serves as a branch prediction hint. The semantics of hint are such that an incorrect branch hint still leads to correct but slower execution. The actual value of hint is allowed to have cache-incoherent mutation during run-time as the dynamic hardware branch-predictor sees fit.

BRNE

BRNE qa, offset

Description:

`BRNE` (branch if not empty) adds the number specified in the `offset` field to the incremented program counter if `qa` is not empty; otherwise, the program counter is just incremented to the next instruction. The data in `qa` is not affected by this instruction. Execution immediately begins at the new `PC` value; there are no branch delay slots.

Operation:

```
if( qa != empty )
     PC ← PC + 1 + offset
else
     PC ← PC + 1
```

Exceptions:

If the destination of the `PC` is in a protected or invalid page, an exception is thrown.

Qualifiers:

The qualifier is ignored by this instruction; `qa` is never dequeued.

Notes:

None.

BREL qa

Description:

BREL (unconditional relative branch) adds the number in qa to the incremented program counter. qa must be of the word type. Execution immediately begins at the new PC value; there are no branch delay slots.

The BREL operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
    PC ← PC + 1 + qa
```

Exceptions:

If the destination of the PC is in a protected or invalid page, an exception is thrown. A type exception is thrown if the type of qa is not word.

Qualifiers:

None.

Notes:

None.

JMP qa, hint

Description:

`JMP` (unconditional jump) sets the value in `PC` to the value in `qa`. Execution immediately begins at the new `PC` value; there are no branch delay slots. `qa` must be of type word.

The `JMP` operation is only executed if `qa` is available and there is no back-pressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
   PC ← qa
```

Exceptions:

If the destination of the `PC` is in a protected or invalid page, an exception is thrown.

Qualifiers:

None.

Notes:

The `hint` field is an implementation-specific 48-bit number that serves as a jump prediction destination hint. The semantics of `hint` are such that an incorrect jump hint still leads to correct but slower execution. The actual value of `hint` is allowed to have cache-incoherent mutation during run-time as the dynamic hardware jump-predictor sees fit.

MOVE

MOVE qa, qc

Description:

`MOVE` (move) takes the value in `qa` and puts it into `qc`. The exact state of the queues after the `MOVE` instruction depends on the @ (copy/clobber) modifiers applied to the queue specifiers.

The `MOVE` operation is only executed if `qa` is available and there is no back-pressure on `qc`. Otherwise, the instruction stalls.

Operation:

`qc` ← `qa`

Exceptions:

An operation exception is thrown if a copy operator is applied to data in `qa` that is tagged non-copyable. The result in `qc` is tagged as invalid, and the original value remains untouched in `qa`.

Qualifiers:

None.

Notes:

Exact semantics vary according to the use of the @ modifier.

MOVECF n, qc

Description:

MOVECF (move floating point constant) takes the 32-bit floating-point constant specified in n, converts it to the nearest ADAM 64-bit floating point number, and puts the properly typed result into qc. The exact state of qc after the MOVECF instruction depends on the @ (copy/clobber) modifier applied to the queue specifier.

The MOVECF operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
qc ← (floating-point) n
type(qc) ← floating-point
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Because of the conversion from a 32-bit opcode-stored representation to a 64-bit standard ADAM floating point representation, the result in qc may exhibit some small roundoff error when compared to the desired constant.

Exact semantics vary according to the use of the @ modifier.

MOVECL

MOVECL n, qc

Description:

MOVECL (move long integer constant) takes the 32-bit constant specified in n, sign-extends it to an ADAM native 64-bit word, and puts the properly typed result into qc. The exact state of qc after the MOVECL instruction depends on the @ (copy/clobber) modifier applied to the queue specifier.

The MOVECL operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
qc ← SEXT(n)
type(qc) ← word
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Exact semantics vary according to the use of the @ modifier.

MOVECI n, qc

Description:

`MOVECI` (move packed integer constant) takes the 32-bit constant specified in `n`, places it in the lower bits of a packed integer, sets the upper bits of the packed integer to zero, and puts the properly typed result into `qc`. The exact state of `qc` after the `MOVECI` instruction depends on the @ (copy/clobber) modifier applied to the queue specifier.

The `MOVECI` operation is only executed if there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
qc.a ← 0
qc.b ← n
type(qc) ← packed integer
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Exact semantics vary according to the use of the @ modifier.

MOVECS

MOVECS n, qc

Description:

MOVECS (move packed short constant) takes the dual 16-bit packed short constant specified in n, places it in the lower bits of a packed short, sets the upper bits of the packed short to zero, and puts the properly typed result into qc. The exact state of qc after the MOVECS instruction depends on the @ (copy/clobber) modifier applied to the queue specifier.

The MOVECS operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
qc.a ← 0
qc.b ← 0
qc.c ← n[31:16]
qc.d ← n[15:0]
type(qc) ← packed short
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Exact semantics vary according to the use of the @ modifier.

MOVECC n, qc

Description:

MOVECC (move packed unicode character constant) takes the dual 16-bit packed unicode character constant specified in n, places it in the lower bits of a packed char, sets the upper bits of the packed char to zero, and puts the properly typed result into qc. The exact state of qc after the MOVECC instruction depends on the @ (copy/clobber) modifier applied to the queue specifier.

The MOVECC operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
qc.a ← 0
qc.b ← 0
qc.c ← n[31:16]
qc.d ← n[15:0]
type(qc) ← packed character
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Exact semantics vary according to the use of the @ modifier.

PACKN	qa, qb, qc, n

Description:

PACKN (Pack Anything) takes the data in qa and inserts it at a position specified by n into the data from qb, and places the result into qc. qa must be of type word, and qb must be of a packed integer type. The result in qc has the same type as qb.

The PACKN operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == word )
   if( type(qb) == packed int )
      if( n == 0 )
         qc.a ← qa & 0xFFFFFFFF
         qc.b ← qb.b
      else
         qc.a ← qb.a
         qc.b ← qa & 0xFFFFFFFF
   elif( type(qb) == packed short or packed char )
      if( n == 0 )
         qc.a ← qa & 0xFFFF
         qc.b ← qb.b
         qc.c ← qb.c
         qc.d ← qb.d
      elif( n == 1 )
         qc.a ← qb.a
         qc.b ← qa & 0xFFFF
         qc.c ← qb.c
         qc.d ← qb.d
      elif( n == 2 )
         qc.a ← qb.a
         qc.b ← qb.b
         qc.c ← qa & 0xFFFF
         qc.d ← qb.d
      else
         qc.a ← qb.a
         qc.b ← qb.b
         qc.c ← qb.c
         qc.d ← qa & 0xFFFF
   else
      throw type exception
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

PACKH

PACKH	qa, qb, qc

Description:

PACKH (Pack High Half of Packed Short or Char) takes packed integer data in qa, masks the data and inserts it into the high half of qb, and places the result into qc. qb must be of type packed short or packed char. The result in qc has the same type as qb.

The PACKH operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == packed int && type(qb) == packed short or packed char)
   qc.a ← qa.a & 0xFFFF
   qc.b ← qa.b & 0xFFFF
   qc.c ← qb.c
   qc.d ← qb.d
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

PACKL

PACKL	qa, qb, qc

Description:

PACKL (Pack Low Half of Packed Short or Char) takes packed integer data in qa, masks the data and inserts it into the low half of qb, and places the result into qc. qb must be of type packed short or packed char. The result in qc has the same type as qb.

The PACKL operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == packed int && type(qb) == packed short or packed char)
   qc.a ← qb.a
   qc.b ← qb.b
   qc.c ← qa.a & 0xFFFF
   qc.d ← qa.b & 0xFFFF
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

PACKI qa, qb, qc

Description:

PACKI (Pack to Packed Integer) takes word data in qa and qb, masks the data and packs it into a packed integer stored in qc.

The PACKI operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa,qb) == word )
   qc.a ← qa & 0xFFFFFFFF
   qc.b ← qb & 0xFFFFFFFF
   type(qc) ← packed integer
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

UNPACK

UNPACK	qa, qb, qc

Description:

`UNPACK` (Unpack) takes a packed integer type `qa` and extracts and sign-extends the data at location `qb` into `qc`. The result `qc` is of type word.

The `UNPACK` operation is only executed if both `qa` and `qb` operands are available and there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
if(type(qb) == word)
    if(type(qa) == packed int)
        if(qb == 0)
            qc ← SEXT(qa.a)
        else
            qc ← SEXT(qa.b)
    elif(type(qa) == packed short or packed char)
        if(qb == 0)
            qc ← SEXT(qa.a)
        elif(qb == 1)
            qc ← SEXT(qa.b)
        elif(qb == 2)
            qc ← SEXT(qa.c)
        else
            qc ← SEXT(qa.d)
    else
        throw type exception
else
    throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

UNPACKC	qa, n, qc

Description:

UNPACKC (Unpack with constant) takes a packed integer type qa and extracts and sign-extends the data at location n into qc. The result qc is of type word.

The UNPACKC operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == packed int)
   if(n == 0)
      qc ← SEXT(qa.a)
   else
      qc ← SEXT(qa.b)
elif(type(qa) == packed short or packed char)
   if(n == 0)
      qc ← SEXT(qa.a)
   elif(n == 1)
      qc ← SEXT(qa.b)
   elif(n == 2)
      qc ← SEXT(qa.c)
   else
      qc ← SEXT(qa.d)
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

FLUSHQ

FLUSHQ qc

Description:

FLUSHQ (Flush Queue) is a special-format instruction, where qc is interpreted as an immediate constant. FLUSHQ discards all values currently in the queue specified by the immediate constant qc. The function of FLUSHQ upon a queue which has mappings to other contexts, be it head or tail mappings, is UNPREDICTABLE. If qc is already empty, nothing happens and execution continues.

Operation:

```
qc ← empty
```

Exceptions:

Throws a mapping exception if qc has any mappings.

Qualifiers:

None.

Notes:

None.

PROCID qc

Description:

`PROCID` (Get Process ID) places the value of the current context ID into `qc`. `qc` is a capability with the owner bit set. In addition, the read and write bits are set. If the context ID is to be passed to another thread, care must be taken to set the permissions properly.

The `PROCID` operation is only executed if there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
qc ← context ID
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

PTRSIZE

PTRSIZE	qa, qc

Description:

PTRSIZE (Get Pointer Size) computes the size of the region of data pointed to by the capability in qa and places the size, in words, in qc. The PTRSIZE operation is valid on any capability, regardless of its permissions. The result in qc is of the word type.

The PTRSIZE operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if( type(qa) == capability )
   qc ← sizeof(qa) in words
   type(qc) ← word
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

CONSUME

CONSUME qa

Description:

CONSUME (Consume Data) reads exactly one piece of data out of qa and discards it. If qa is initially empty, CONSUME blocks.

Operation:

```
while( qa is empty )
   stall
if( no @ operator on qa )
   dequeue head of qa
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

SEMPTY

SEMPTY	qa, qc

Description:

`SEMPTY` (Set if Empty) is a special format instruction, where `qa` is interpreted as an immediate constant. `SEMPTY` tests to see if the queue specified by the immediate constant `qa` is empty, and if it is, it places an integer 1 into `qc`. Otherwise, a 0 is written into `qc`. The type of the result `qc` is word.

Operation:

```
if((qa & 0x7F) is empty)
   qc ← 1
else
   qc ← 0
type(qc) ← word
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

EEQ

EEQ qa

Description:

`EEQ` (forcE Empty Queue) is a special format instruction, where `qa` is interpreted as an immediate constant. `EEQ` tests to see if the queue specified by the immediate constant `qa` is empty, and if it is, it increments the PC; if not, the PC remains constant and a yielding stall is reported to the scheduler.

Operation:

```
if((qa & 0x7F) is empty)
   pc ← pc + 1
else
   pc ← pc
```

Exceptions:

None.

Qualifiers:

None.

Notes:

This instruction complicates the implementation of the processor core. An alternative would be to use SEMPTY and a BRZ instruction to create a programmatic loop to check for the emptiness of a queue. However, for the purposes of backward compatibility with an older ISA, it is included in the documentation.

RANDOM

RANDOM qc

Description:

RANDOM (Generate Random Number) places a cryptographically secure random integer of type word into qc. RANDOM may be implemented as an external hardware device to the processor. Because 64 bits of entropy must be collected for each RANDOM instruction, it is possible to request random numbers faster than the processor or device is capable of generating them. In this case, the operation blocks until a random number becomes available. In order to smooth out demand patterns, the number generating device may elect to queue up several pre-generated numbers.

The RANDOM operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
qc ← random number between −2^63 and 2^63 − 1
type(qc) ← word
```

Exceptions:

None.

Qualifiers:

None.

Notes:

The exact implementation of the RANDOM function should be disclosed in a public fashion before it can be trusted. More information on cryptographically secure random numbers can be found in Annex D.6 “Random number generation” of the IEEE 1363-2000 standard and in RFC1750, “Randomness Recommendations for Security”. A user desiring to verify the randomness properties of the RANDOM instruction may wish to refer to Ueli M. Maurer’s “A Universal Statistical Test for Random Bit Generators”, *Institute of Theoretical Computer Science, ETH Zürich*, 1992, Journal of Cryptology, Vol. 5, No. 2.

GETSTAT

GETSTAT	qc

Description:

`GETSTAT` (Get Status Register) copies the contents of the status register into `qc`. There are some portions of the status register that are implementation-specific. `qc` is of type word.

The `GETSTAT` operation is only executed if there is no backpressure on `qc`. Otherwise, the instruction stalls.

Operation:

```
qc ← status register
type(qc) ← word
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Please refer to the implementation notes and the architecture specification for the meaning of the status register bits.

SETSTAT

SETSTAT qa

Description:

SETSTAT (Set Status Register) copies the contents of qa into the modifiable portions of the status register. There are some portions of the status register that are implementation-specific. qa must be of type word.

The SETSTAT operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == word)
   status register ← qa
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

Please refer to the implementation notes and the architecture specification for the meaning of the status register bits. Some of the bits of the status register are read-only and are unaffected by SETSTAT.

GETEX

GETEX qc

Description:

GETEX (Get Exception Context ID) places the current exception handler's context ID into qc. The permissions on the exception handler ID are set to opaque and owner.

The GETEX operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
qc ← Exception Register
type(qc) ← capability
permissions(qc) ← opaque, owner
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

SETEX

SETEX qa

Description:

SETEX (Set Exception Context ID) sets the current context's exception handler ID to be the capability in qa. The operation blocks if qa is applying backpressure.

Operation:

```
if(type(qa) == capability)
   Exception Register ← qa
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

None.

THROW

Description:

THROW (Throw Soft Exception) causes the current context to be set to the exception handler context and for the PC to jump to the exception handler's server code. In addition, the current context ID is saved into the Exceptioned Context ID register. The user may layer additional conventions on top of the basic THROW semantics; for example, the user may require that q127 contain a soft exception ID.

Operation:

```
PC ← PC + 1
Exceptioned Context ID ← context ID
context ID ← exception handler ID
PC ← exception handler server code start
```

Exceptions:

None.

Qualifiers:

None.

Notes:

Note that there is no requirement for a saved PC because the PC of the exceptioned context is not overwritten by the exception handler PC: the context ID is set to the exception handler before the PC is modified.

This is a multi-cycle, variable execution duration instruction.

EXTAG	qa, qc

Description:

EXTAG (Extract Tag) extracts the tag bits out of qa and places them into qc. The tag bits are placed in the MSB's of qc and zero-padded to the right. The tag region of a piece of data includes the top 16 bits, whereas the tag region for a capability includes the top 45 bits. The type of the result in qc is word.

The EXTAG operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == capability)
   qc ← {qa[79:55],39'b0}
else    qc ← {qa[79:64],48'b0}
type(qc) ← word
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

SETTAG

SETTAG	qa, qb, qc

Description:

SETTAG (Set Tag) sets the tag of the data in qb to the value of the LSB's of qa, and places the result into qc. This is a very powerful operator, as it can force a literal binary transmutation of data types and change several important attributes about a piece of data. If the value of the bits in qa corresponds to a capability, the type of qb must also be a capability, and the owner bit for qb must be set. qa must be of type word.

The SETTAG operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == word)
   if(type(qb) == capability)
      if(!owner(qb))
         throw operation exception
      else
         tags(qb) ← qa[63:39]
   else
      if(qa[63] == 1)
         throw operation exception
      else
      tags(qb) ← qa[63:48]
else
   throw type exception
```

Exceptions:

Operation exception, type exception.

Qualifiers:

None.

Notes:

None.

ALLOCATE qa, qb, qc

Description:

ALLOCATE (Allocate Capability) creates a capability qc of the size nearest to the number of words specified in qb. The address of the capability and the increment-only bit are set to restrict the accessible portion of the capability to exactly the size specified in qb. qb must be of type word. If the allocation fails, qc is returned as an invalid capability, and an out of memory exception is thrown. qa contains an allocation metric that guides where the allocated memory should be placed in the system. qa must be of type packed char or a capability. If qa is a capability, the system attempts to allocate the new capability close to the capability in qa.

The ALLOCATE operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qb) == word && (type(qa) == packed char || (type(qa) == capability)))
   if(qa words available)
      qc ← capability of size qa bytes
   else
      qc ← invalid capability
      throw out of memory exception
else
   throw type exception
```

Exceptions:

Out of memory exception, type exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete. This instruction is a “lazy” instruction.

The format of the allocation metric is implementation dependant. The current implementation scheme calls for the packed char to contain the following sixteen-bit char values, from MSB to LSB: ignored, ignored, expected communication frequency, desired latency.

ALLOCATEC qa, n, qc

Description:

ALLOCATEC (Allocate Capability, Size in Constant Field) creates a capability qc of the size nearest to the number of words specified in n. The address of the capability and the increment-only bit are set to restrict the accessible portion of the capability to exactly the size specified in n. If the allocation fails, qc is returned as an invalid capability, and an out of memory exception is thrown. qa contains an allocation metric that guides where the allocated memory should be placed in the system. qa must be of type packed char or of type capability. If qa is a capability, the system attempts to allocate the new capability close to the capability in qa.

The ALLOCATEC operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == packed char || type(qa) == capability)
   if(n words available)
      qc ← capability of size n bytes
   else
      qc ← invalid capability
      throw out of memory exception
else
   throw type exception
```

Exceptions:

Out of memory exception, type exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete. This instruction is a “lazy” instruction.

The format of the allocation metric is implementation dependant. The current implementation scheme calls for the packed char to contain the following sixteen-bit char values, from MSB to LSB: ignored, ignored, expected communication frequency, desired latency.

MML qa, qb

Description:

MML (Map Memory Load) maps the queue number specified in qa to a load address queue, and maps the return data of the load into the queue number specified in qb. qa and qb must be of type word.

The memory subsystem expects that the first address entered into a memory address queue be the access capability, and that subsequent entries to the load address queue be offsets on the initial capability. Enqueueing the initialization capability does not cause the memory subsystem to return a load value. If a capability is sent to the memory subsystem following the initialization capability, the new capability subsumes the old one; again, no load value is returned in response to this load capability being sent.

This operation stalls until both qa and qb contain a value.

Operation:

```
if(type(qa,qb) == word)
   MAP (qa & 0x7F) to memory load address queue
   MAP memory load return data queue to (qb & 0x7F)
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete the mapping, but the PC is allowed to increment in one cycle. This does not lead to incorrect operation unless the user unmaps the memory mapping instruction and then immediately re-maps the memory mapping. Users should avoid unmapping and remapping memory maps using the same queues within the same context. Note that it is perfectly safe to re-initialize an existing memory mapping by sending a new capability to the address queue.

When unmapping a memory mapped queue pair, the user is responsible for unmaping both the address and the data queue. There is nothing fundamentally incorrect about unmapping one queue only; however, it may lead to confusion if the queue mapping is re-used, and the garbage collector will not de-allocate memory that has even a partial mapping to its capability.

MMS qa, qb

Description:

MMS (Map Memory Store) maps the queue number specified in qa to a store address queue, and maps the queue number specified in qb to a store data queue. qa and qb must be of type word.

Data and addresses may be enqueued at differing times and rates, but the invariant is that the store blocks until both queues have at least one element in them, and that data and address pairs are strictly correlated by their relative order in the queues.

The memory subsystem expects that the first address entered into a memory address queue be the access capability; this first access is **not** matched with a data element in the store data queue. Subsequent addresses are then interpreted as offsets to the initial access capability and are paired with data values in the store data queue.

This operation stalls until both qa and qb contain a value.

Operation:

```
if(type(qa,qb) == word)
   MAP (qa & 0x7F) to memory store address queue
   MAP (qb & 0x7F) to memory store data queue
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete the mapping, but the PC is allowed to increment in one cycle. This does not lead to incorrect operation unless the user unmaps the memory mapping instruction and then immediately re-maps the memory mapping. Users should avoid

unmapping and remapping memory maps using the same queues within the same context. Note that it is perfectly safe to re-initialize an existing memory mapping by sending a new capability to the address queue.

When unmapping a memory mapped queue pair, the user is responsible for unmaping both the address and the data queue. There is nothing fundamentally incorrect about unmapping one queue only; however, it may lead to confusion if the queue mapping is re-used, and the garbage collector will not de-allocate memory that has even a partial mapping to its capability.

EXCH	qa, qb, qc

Description:

EXCH (Declare Exchange Tuple) marks the queues numbers specified in qa, qb and qc as a memory exchange tuple. qa is set to be the address queue, qb is set to be the data in queue, and qc is set to be the data out queue. All of qa, qb, and qc are interpreted to be immediate constants. The exchange tuple must be initialized by moving a capability into qa prior to moving an address offset into qa.

Once the tuple has been initialized with an address value, the next piece of data moved into qb is exchanged atomically with the contents of memory at the specified address, and the contents of the memory location prior to the exchange is placed in qc.

This operation is guaranteed by the memory system to be atomic at the memory side; however, no other relative timings are guaranteed.

The EXCH mapping remains in effect until it is undone with an UNMAPQ instruction. The user must unmap all three mappings.

Operation:

```
MAP qa to atomic memory address queue
MAP qb to atomic memory incoming data queue
MAP qc to atomic memory return data queue
```

Exceptions:

Type exception and exchange exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

SPAWN	qa, qb, qc

Description:

SPAWN (Spawn) starts a new thread by allocating space for the thread, creating an entry in the thread scheduler for the thread with PC set to the value in qb, and returning the thread ID (which is also a capability to thread's data) in qc. The permissions of the thread ID capability are set to opaque and not owner. qa contains a spawning metric that is used to guide the run-time as to where the thread should be spawned. If qa is a capability, the system attempts to allocate the new thread close to the capability in qa.

qa must be of type packed char or type word, and qb must be of type word.

The SPAWN operation is only executed if qa is available and there is no back-pressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qb) == word && (type(qa) == packed char || (type(qa) == capability)))
    qc ← new thread capability
    if(qc == invalid)
        throw out of memory exception
    else
        create thread scheduler entry (new thread ID, PC = qa)
else
    throw type exception
```

Exceptions:

Type exception, Out of memory exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete. This is a "lazy" instruction.

The format of the spawning metric is implementation dependant. The current implementation scheme calls for the packed char to contain the following sixteen-bit char values, from MSB to LSB: expected children, memory requirement, computation requirement, desired latency.

SPAWNL qa, qb, qc

Description:

SPAWNL (Load Code and Spawn) starts a new thread by allocating space for the thread, loading its code specified in qb into code space, and creating an entry in the thread scheduler for the thread with PC set to the value in qa, and returning the thread ID (which is also a capability to thread's data) in qc. The permissions of the thread ID capability are set to opaque and not owner. The size of the space to be allocated for the thread is encoded in an OSIZE opcode that should be the first instruction of the new thread.

qa must be of type word, and qb must be a capability to a character array that describes a universal locator for the code resource.

The SPAWNL operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == word && type(qb) == capability)
    load code specified by qb into code space
    qc ← capability of size indicated in OSIZE opcode at address in qa
    if(qc == invalid)
        throw out of memory exception
    else
        create thread scheduler entry (new thread ID, PC = qa)
else
    throw type exception
```

Exceptions:

Type exception, Out of memory exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

SPAWNC qa, n, qc

Description:

SPAWNC (Spawn with PC-constant offset) starts a new thread by allocating space for the thread, creating an entry in the thread scheduler for the thread with PC set to the value of PC + 1 + n, and returning the thread ID (which is also a capability to thread's data) in qc. The permissions of the thread ID capability are set to opaque and not owner. The size of the space to be allocated for the thread is encoded in an OSIZE opcode that should be the first instruction of the new thread. qa contains a spawning metric that is used to guide the run-time as to where the thread should be spawned. If qa is a capability, the system attempts to allocate the new capability close to the capability in qa.

The SPAWNC operation is only executed if there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == packed char || type(qa) == capability)
   qc ← capability of size in OSIZE opcode at (n + PC + 1)
   if(qc == invalid)
      throw out of memory exception
   else
      create thread scheduler entry (new thread ID, PC = n + PC + 1)
else
   throw type exception
```

Exceptions:

Out of memory exception and type exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete. This is a "lazy" instruction.

The format of the spawning metric is implementation dependant. The current implementation scheme calls for the packed char to contain the following

sixteen-bit char values, from MSB to LSB: expected children, memory requirement, computation requirement, desired latency.

MAPQ

MAPQ	qa, qb, qc

Description:

MAPQ (Map Queue) is a special-format instruction. qa is actually interpreted as an immediate constant: it specifies the queue number in the current context that is to be mapped. MAPQ does not actually read or modify the contents of qa in any way. The copy/clobber modifier has no effect on the value of qa in this case. qb specifies the queue number to read for the queue number of the destination mapping, and qc specifies the queue number to read for the destination context ID.

The MAPQ operation is only executed if both qb and qc operands are available.

Operation:

```
if(type(qb) == word && type(qc) == capability)
   map queue ``qa''.tail in current context to
   queue ((qb & 0x7F) ≫ 7).head in context qc
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

The odd format of this instruction is an artifact of backward compatibility with an earlier version of the instruction set. This instruction may be represented inside the hardware implementation in a more typical fashion and require the assembler to do a simple format translation. This instruction may take multiple cycles to complete.

MAPQC

MAPQC	qa, qb, qc

Description:

MAPQC (Map Queue with Destination as Constant) is a special-format instruction. qa and qb are actually interpreted as immediate constants: they specify the queue number in the current context and the destination queue number, respectively, that is to be mapped. MAPQC does not actually read or modify the contents of qa or qb in any way. The copy/clobber modifier has no effect on the value of qa and qb in this case. qc specifies the queue number to read for the destination context ID.

The MAPQC operation is only executed if the qc operand is available.

Operation:

```
if(type(qc) == capability)
   map queue ``qa''.tail in current context to
   queue ``qb''.head in context qc
else
   throw type exception
```

Exceptions:

Type exception.

Qualifiers:

None.

Notes:

The odd format of this instruction is an artifact of backward compatibility with an earlier version of the instruction set. This instruction may be represented inside the hardware implementation in a more typical fashion and require the assembler to do a simple format translation. This instruction may take multiple cycles to complete.

MAPSQ

MAPSQ	qa, qb

Description:

`MAPSQ` (Map Queue Source) is a special-format instruction. `qa` and `qb` are actually intepreted as immediate constants. `MAPSQ` creates a mapping such that every element enqueued *by the network interface* into the queue specified in the immediate constant `qa` also enqueues the context ID of the data's source into the queue specified by the immediate constant `qb`. The arrival of data from the network interface in the queue specified by `qa` is guaranteed to be simultaneous with the arrival of the context ID in the queue specified by `qb`. The resulting type of the IDs in `qb` are capability, with the opaque bit set and the owner bit cleared.

Operation:

```
map incoming data source ID of queue (qa & 0x7F) to (qb & 0x7F)
```

Exceptions:

None.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

Note that data arriving in `qa` via local operations do not cause `qb` to have the source enqueued; thus, it is not recommended to share `qa` as both a target for local and remote operations.

MAPDROP

MAPDROP qa

Description:

`MAPDROP` (Set Mapping to Drop Mode) is a special-format instruction where `qa` is interpreted as an immediate constant. `MAPDROP` sets the mode of the mapping of the queue number specified by the immediate constant `qa` to "drop" mode. In this mode, backpressure on the queue causes data to be dropped instead of stalling the context. This is particularly useful when implementing pure streaming operators on real-time datatypes such as video or audio.

Operation:

```
set mode of queue (qa & 0x7F) to drop mode
```

Exceptions:

None.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

UNMAPQ qa

Description:

UNMAPQ (Unmap A Queue) is a special format instruction, in that qa is interpreted as an immediate constant. UNMAPQ resets the mapping of the queue specified by the immediate constant qa to the default (current context ID). Care should be taken to guarantee that the specified queue is empty before issuing this instruction, otherwise left-over data that may be in the queue when this instruction retires will never be delivered to its destination.

Operation:

```
set the mapping of queue (qa & 0x7F) to the current context ID
```

Exceptions:

None.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

When unmapping a memory mapped queue pair, the user is responsible for unmaping both the address and the data queue. There is nothing fundamentally incorrect about unmapping one queue only; however, it may lead to confusion if the queue mapping is re-used, and the garbage collector will not de-allocate memory that has even a partial mapping to its capability.

PARCEL qa, qb, qc

Description:

PARCEL (Parcel out a Capability) takes a capability in qa and attempts to create a sub-capability with the address and tags described in qb. The result is placed in qc. qa must be a capability, qb is a word type, and the result qc is a capability. The format of the sub-capability address and tag specifier is 15 bits of tags followed by a 1 bit increment-only field, followed by a 35 bit address field. The unused bits to the left are ignored.

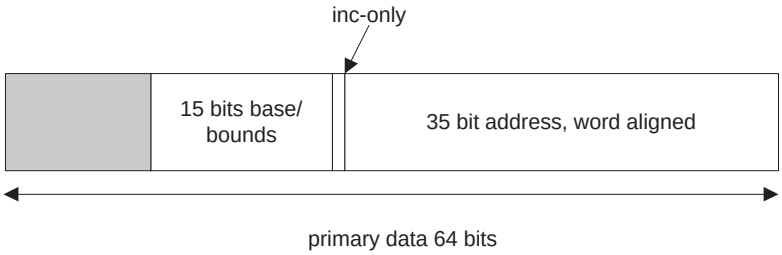


Figure D.1: qb format for the PARCEL instruction

If the capability described by qb is outside the bounds of the given capability in qa, an operation exception is thrown and the result in qc is invalid.

The PARCEL operation is only executed if both qa and qb operands are available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == capability && type(qb) == word)
   qc ← sub-capability of qa described by qb
else
   throw type exception
```

Exceptions:

Type exception and operation exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

MSYNC

MSYNC

Description:

`MSYNC` (Memory Synchronize) causes the current thread to stall until all of the current thread's pending memory operations have completed.

Operation:

```
if(current thread has pending memory operations)
   PC ← PC
   signal structural stall to thread scheduler
else
   PC ← PC + 1
```

Exceptions:

None.

Qualifiers:

None.

Notes:

This instruction will take a variable number of cycles to complete.

LDCODE qa, qc

Description:

LDCODE (Dynamically Load Code) takes a capability in qa which contains a character array that names a code object and its path, attempts to load it into code memory, and returns the absolute PC address of the code as a word in qc. A failure to complete this operation causes a code load exception to be thrown and qc to be invalid.

(Need to determine if the return should be a PC value, or if it should be a context ID to an object server that was started...)

The LDCODE operation is only executed if qa is available and there is no backpressure on qc. Otherwise, the instruction stalls.

Operation:

```
if(type(qa) == capability)
   if(qa.permissions == read, not opaque, valid)
      load code described by character array in qa
      qc ← PC of code entry point
      if(tags(qc) == invalid)
         throw code load exception
   else
      throw operation exception
else
   throw type exception
```

Exceptions:

Type exception, operation exception, and code load exception.

Qualifiers:

None.

Notes:

This instruction may take a variable number of cycles to complete.

OSIZE n

Description:

OSIZE (Object Size Directive) is a compiler directive that uses the “hint” opcode format to inform ADAM how large a region needs to be allocated for a particular thread object. The size of the region to allocate in words is indicated in n. This opcode may be located anywhere, but it only has meaning when it is in the entry point instruction sequence for an object’s initializer code. When executed, this instruction does nothing to the machine state except increment the PC.

Operation:

```
PC = PC + 1
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

HINT

HINT t,hint

Description:

`HINT` (Compiler Hint) is a hint from the compiler or programmer to the ADAM runtime system. A `HINT` instruction has no effect on the ADAM machine state except for incrementing the `PC`; however, it may have a profound impact upon the OS and/or management coprocessor.

The type of hint is encoded in the `t` field, and the actual value of the hint is encoded in the `hint` field. The valid hint types are TBD, but they fall into two broad categories: machine specific and machine independent. Machine specific hints include data placement directives. Machine independent hints include thread swap hints, prefetch directives, and migration hints. A hint with an unrecognized hint type is ignored.

An incorrect hint never leads to incorrect program results; an incorrect just leads to poor performance.

Operation:

```
PC = PC + 1
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

NOP

NOP

Description:

`NOP` (No Operation) A `NOP` instruction has no effect on the ADAM machine state except for incrementing the `PC`.

Operation:

```
PC = PC + 1
```

Exceptions:

None.

Qualifiers:

None.

Notes:

None.

Bibliography

[AB93] Arvind and Stephen Brobst. The evolution of dataflow architectures: from static dataflow to P-RISC. *International Journal of High Speed Computing*, 5(2), 1993. World Scientific Publishing Company.

[ABC+95] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.H. Lim, K. Mackenzie, and D. Yeung. The MIT alewife machine: Architecture and performance. In *Proceedings of the 22^{nd} International Symposium on Computer Architecture*, pages 2–13, Santa Margherita Ligure, Italy, June 1995.

[AHKB00] Vikas Agarwal, M.S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27^{th} Annual International Symposium on Computer Architecture*. ACM, 2000.

[AKK+95] Gail Alverson, Simon Kahan, Richard Korry, Cathy McCann, and Burton Smith. Scheduling on the Tera MTA. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, number 949 in LNCS, pages 19–44, 1995.

[ALKK91] A. Agarwal, B.H. Lim, D. Kranz, and J. Kubiatowicz. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of ASPLOS IV*, pages 224–234, April 1991.

[ASHH88] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15^{th} International Symposium on Computer Architecture*, pages 28–289, June 1988.

[BEY98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[BGKH00] Jeremy H. Brown, J.P. Grossman, Tom Knight, and Andrew Huang. A capability representation with embedded address and near-exact object bounds. Technical Report Project Aries Tech Note 5, Massachusetts Institute of Technology AI Lab, http://www.ai.mit.edu/projects/aries, 2000.

[BL94] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35^{th} Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, November 1994.

[BP92] Jeff Baxter and Janak H. Patel. Profiling based task migration. In *Sixth International IEEE Parallel Processing Symposium*, pages 192–195, 1992.

[Bro02] Jeremy Hanford Brown. *Sparsely Faceted Arrays: A Mechanism Supporting Parallel Allocation, Communication, and Garbage Collection*. PhD thesis, Massachusetts Institute of Technology, 2002.

[CCH+00] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, Jon Wawrzynek, and André DeHon. Stream computations organized for reconfigurable execution (SCORE). In *Conference on Field Programmable Logic and Applications*. Springer-Verlag, August 2000.

[CDV+94] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of ASPLOS VI*, San Jose, CA, 1994.

[Che] Erik Cheever. Runge-kutta methods. http://www.swarthmore.edu/natsci/echeeve1/Ref/ NumericInt/RK2.html.

[CI00a] International Roadmap Committee and ITWGs. International roadmap for semiconductors update. Technical report, SIA, http://public.itrs.net/, 2000.

[CI00b] International Roadmap Committee and ITWGs. International roadmap for semiconductors, PIDS update. Technical report, SIA, http://public.itrs.net/, 2000.

[CI00c] International Roadmap Committee and ITWGs. International roadmap for semiconductors, design update. Technical report, SIA, http://public.itrs.net/, 2000.

[CJDM01] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. High-performance DRAMs in workstation environments. *IEEE Transactions on Computers*, 50(11):1133–1153, November 2001.

[CKD94] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of ASPLOS VI*, October 1994.

[CKW96] Oliver Ciupke, Dietmar Kottmann, and Hans-Dirk Walter. Object migration in non-monolithic distributed applications. In *Proc. of the 16th International Conference on Distributed Computing Systems*, pages 529–536, 1996.

[CM97] Haines D. Cronk and P.M. Mehrotra. Thread migration in the presence of pointers. In *Proceedings of the 30th Hawaii International Conference on System Sciences*, volume 1, pages 292–298. IEEE, 1997.

[Cora] MoSys Corporation. TSMC 0.13μm process fast 1-T SRAM summary. http://www.mosys.com/1t_sram.html. Registration required to access design materials.

[Corb] Taiwan Semiconductor Manufacturing Corporation. 0.18 μm process summary. http://www.tsmc.com/technology/cl018.html.

[CS99] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

[CSS+91] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proc. of 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa-Clara, CA, April 1991.

[DeH90] André DeHon. Fat-tree routing for transit. S.B. thesis AI Technical Report 1224, MIT Artificial Intelligence Laboratory, 1990.

[DeH93] André DeHon. Robust, high-speed network design for large-scale multiprocessing. S.M. thesis AI Technical Report 1445, MIT Artificial Intelligence Laboratory, 200 Technology Square, Cambridge, MA 02139, 1993.

[DS90] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Software Engineering*, 16(6):660–673, 1990.

[ea92] H. Burkhart III et al. Overview of the KSR1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[FKD+95] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine multicomputer. In *Proceedings of the 28^{th} Annual International Symposium on Microarchitecture*, pages 146–156. IEEE, 1995.

[FNN93] Matthew K. Farrens, Pius Ng, and Phil Nico. A comparison of superscalar and decoupled access/execute architectures. In *Proceedings of the 26^{th} Annual International Symposium on Microarchitecture*, pages 100–103. IEEE, 1993.

[Gal96] Mike Galles. The SGI spider chip. In *Proceedings of Hot Interconnects IV*, pages 141–146. IEEE, 1996.

[GB02] J.P. Grossman and Jeremy Brown. A lightweight idempotent messaging protocol for faulty networks. In *Appearing in the*

Proceedings of the 2002 Symposium on Parallel Algorithms and Architectures, 2002.

[GBHK00] J.P. Grossman, Jeremy Brown, Andrew Huang, and Tom Knight. Using SQUIDs to address forwarding pointer aliasing. Technical Report Project Aries Tech Note 4, Massachusetts Institute of Technology AI Lab, http://www.ai.mit.edu/projects/aries, 2000.

[GHI94] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: The Terasys massively parallel PIM array. *IEEE Computer*, 28(4):23–31, 1994.

[Gro01] J.P. Grossman. Design and evaluation of the Hamal parallel computer. Doctoral research proposal, Massachusetts Institute of Technology, 2001.

[GWM90] A Gupta, W.D. Weber, and T Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Proceedings of the International Conference on Parallel Processing*, pages 312–321, August 1990.

[Har00] Shawn R. Hartsock. Gravity works! http://modzer0.cs.uaf.edu/ hartsock/C_Cpp/OpenGL/Gravity.html, March 2000. Originally found on www.planet-source-code.com.

[HHK+01] Joseph Hall, Jason Hartline, Anna R. Karlin, Jared Saia, and John Wilkes. On algorithms for efficient data migration. In *Twelfth Annual Symposium on Discrete Algorithms*, pages 620–629. ACM SIGACT and SIAM, 2001.

[HHS+00] Lance Hammond, Ben Hubbert, Michael Siu, Manohar Prabhu, Mike Chen, and Kunle Olukotun. The stanford hydra CMP. *IEEE Micro Magazine*, March–April 2000.

[HS94a] Chao-Ju Hou and Kang G. Shin. Design and evaluation of effective load sharing in distributed real-time systems. *IEEE Transactions on Parallel Distributed Systems*, 5(7):704–719, July 1994.

[HS94b] Chao-Ju Hou and Kang G. Shin. Load sharing with consideration of future task arrivals in heterogeneous distributed real-time systems. *IEEE Transactions on Computers*, 43(9):1076–1090, September 1994.

[Hsi95] Wilson Cheng-Yi Hsieh. *Dynamic Computation Migration in Distributed Shared Memory Systems*. Ph.d. thesis, MIT, 1995.

[HSU+01] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1 2001. http://developer.intel.com/technology/itj/q12001/articles/art_2.htm.

[HWW93] Wilson C. Hsieh, Paul Wang, and William E. Weihl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proceedings of the Fourth ACM PPOPP*, pages 239–248, California, May 1993.

[Ian88] Robert Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15^{th} Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, May 1988.

[Inc01] Object Management Group Inc., editor. *The Common Object Request Broker: Architecture and Specification (Version 2.5)*. http://www.omg.org, September 2001.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Joe96] C. F. Joerg. The cilk system for parallel multithreaded computing. Technical Report MIT/LCS/TR-701, MIT Laboratory for Computer Science, 1996.

[KMSM01] V. Kalogeraki, P.M. Melliar-Smith, and L.E. Moser. Dynamic migration algorithms for distributed object systems. In *21st IEEE International Conference on Distributed Computing Systems*, pages 119–126, Phoenix, Arizona, April 2001.

[KPP+97] C. Kozyrakis, S. Perissakis, D. Patterson, T. Andreson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, pages 75–78, September 1997.

[LBCF+00] J.M. London, E. Barkley, Jr. C.G. Fonstand, A. Loomis of MIT Lincoln Laboratory, and Fari Assaderaghi of IBM. Silicon-on-gallium arsenide for optoelectronic integration. Technical report, MIT MTL, http://www-mtl.mit.edu/mtlhome/6Res/AR2000/AR00index.html, 2000.

[LBF+98] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a

RAW machine. In *Proceedings of ASPLOS-VIII, California*. ACM, 1998.

[LFA96] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. *Journal of Parallel and Distributed Computing*, 37(1):41–54, August 1996.

[LH94] Ben Lee and A.R. Huron. Dataflow architectures and multithreading. *IEEE Computer*, August 1994.

[LJ90] Carl E. Love and Harry F. Jordan. An investigation of static versus dynamic scheduling. In *Proceedings of the 17^{th} Annual International Symposium on Computer Architecture*, pages 192–201. IEEE, 1990.

[LL97] James Laudon and Daniel Lenoski. System overview of the SGI Origin 200/2000 product line. In *COMPCON*. IEEE, 1997.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, and J. Hennessy. The stanford dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[LW95] Daniel E. Lenoski and Wolf-Dietrich Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann, 1995.

[Mac00] International Business Machine. Blue Logic Cu-11 ASIC. Technical Report SA14–2451–00, IBM, 2000.

[Mar00] Norman Margolus. An embedded DRAM architecture for large-scale spatial-lattice computations. In *The 27^{th} Annual International Symposium on Computer Architecture*, pages 149–160. IEEE, 2000.

[McF97] Grant W. McFarland. *CMOS Technology Scaling and Its Impact on Cache Delay*. PhD thesis, Stanford University, June 1997.

[MJC+99] C.E. Molnar, I.W. Jones, W.S. Coates, J.K. Lexau, S.M. Fairbanks, and I.E. Sutherland. Two FIFO ring performance experiments. In *Proceedings of the IEEE*, volume 87, pages 297–307, February 1999.

[MJCL97] C.E. Molnar, I.W. Jones, W.S. Coates, and J.K. Lexau. A FIFO ring performance experiment. In *Proceedings of the Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 279–289, 1997.

[MPJ+00] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 13^{th} annual international symposium on computer architecture*, June 2000.

[MSAD90] William Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. The effects of memory latency and fine-grained parallelism on astronautics ZS-1 performance. In *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences*, volume 1, pages 288–296. IEEE, 1990.

[MSAD91] William Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. Architectural vs. delivered performance of the IBM RS/6000 and the Astronautics ZS-1. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, volume 1, pages 397–408. IEEE, 1991.

[MSW93] Henk L. Muller, Paul W.A. Stallard, and David H.D. Warren. The data diffusion machine with a scalable point-to-point net-

work. Technical Report CSTR-93-17, University of Bristol Computer Science Department, October 1993.

[NA89] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, May 1989.

[ND91] Peter R. Nuth and William J. Dally. A mechanism for efficient context switching. In *International Conference on Computer Design*, pages 301–304, 1991.

[ND95] Peter R. Nuth and William J. Dally. The named-state register file: Implementation and performance. In *Proceeding of the First IEEE Symposium on High-Performance Computer Architecture*, pages 4–13, 1995.

[NWD93] M.D. Noakes, D.A. Wallach, and W.J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proceedings of the 20th Annual Symposium on Computer Architecture*, pages 224–235, 1993.

[OCS98] M. Oksin, F.T. Chong, and T. Sherwood. Active pages: A computational model for intelligent memory. In *The 25th Annual Symposium on Computer Architecture*, pages 192–203. IEEE, 1998.

[ONH+96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of ASPLOS-VII, Cambridge MA*. ACM, 1996.

[PBB93] G. A. Papadopoulos, Greiner R. Boughton, and M.J. Beckerle. *T: Integrated building blocks for parallel computing. In *Proceedings of the Conference on Supercomputing*, pages 623–635, 1993.

[PM83] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the 9^{th} ACM Symposium on Operating Systems Principles*, pages 110–119, New York, NY, 1983. ACM Press.

[RC96] Ellard T. Roush and Roy H. Campbell. Fast dynamic process migration. In *Proceedings of the 16^{th} International Conference on Distributed Computing Systems*, pages 637–645. IEEE, 1996.

[RSAU91] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, 1991.

[Sch] Wayne Schlitt. Xstar n-body solver. http://www.midwestcs.com/xstar/.

[Sco96] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of ASPLOS VII*, Massachusetts, 1996. ACM.

[SDV+87] J.E. Smith, G.E. Dermer, B.D. Vanderwarn, S.D. Klinger, C.M. Rozewski, D.L. Folwler, K.R. Scidmore, and J.P. Laudon. The ZS-1 central processor. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–204, October 1987.

[Sem] Matrix Semiconductor. Matrix semiconductor website. http://www.matrixsemi.com.

[SHK95] Behrooz A. Shirazi, Ali R. Hurson, and Krishna M. Kavi, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[SKA01] Michael Sung, Ronny Krashinsky, and Krste Asanović. Multithreading decoupled architectures for complexity-effective general purpose computing. In *Workshop on Memory Access Decoupled Architectures, PACT-01*, Barcelona, Spain, September 2001.

[Smi82a] Burton Smith. Architecture and applications of the HEP multiprocessor computer system. In *Proceedings of the International Society for Optical Engineering*, pages 241–248, 1982.

[Smi82b] James E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9^{th} Annual International Symposium on Computer Architecture*, Austin, Texas, April 1982.

[Ste85] David Stevenson. IEEE standard for binary floating-point arithmetic. ANSI/IEEE standard 754-1985, August 1985.

[TP96] Josep Torrellas and David Padua. The illinois aggressive coma multiprocessor project (I-ACOMA). In *6^{th} Symposium on the Frontiers of Massively Parallel Computing*, October 1996.

[Tuc84] D.B. Tuckerman. Heat-transfer microstructures for integrated circuits. Ph.D. dissertation, Stanford University, 1984.

[Van02] Benjamin Mead Vandiver. Compilation to a queue-based architecture. Master's thesis, Massachusetts Institute of Technology, 2002.

[vCGS92] T. vonEicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation, 1992.

[VG98] Alexander V. Veidenbaum and K. A. Gallivan. Decoupled access DRAM architecture. *IEEE Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 94–103, 1997,1998.

[WC01] Robert Woods-Corwin. A high-speed fault-tolerant interconnect fabric for large-scale multiprocessing. Master's thesis, Massachusetts Instituted of Technology, 2001.

[WCD+95] Chih-Po Wen, Soumen Chakrabarti, Etienne Deprit, Arvind Krishnamurthy, and Katherine Yelick. Runtime support for portable distributed data structures. In *Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, May 1995.

[WGQH98] B. Weissman, B. Gomes, J. Quittek, and M. Holtkamp. Efficient fine-grain thread migration with active threads. In *Submitted to the 12^{th} International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, pages 410–414, 1998.

[Wul92] William A. Wulf. Evaluation of the WM architecture. In *Proceedings of the 19^{th} Annual International Symposium on Computer Architecture*, pages 382–390. ACM, 1992.

[XL97] Chengzhong Xu and Francis C.M. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, 1997.