Q-Machine: A Spatially-Aware Decentralized Computer Architecture

Proposal for Research in
Partial Fulfillment of the Requirements for a Doctorate of Philosophy in
Electrical Engineering and Computer Science at the
Massachusetts Institute of Technology

Andrew "bunnie" Huang

April 25, 2001

Table of Contents

Table of Figures

Table of Tables

1 Introduction

The field of computer architecture is in a constant state of flux. The furious pace of progress in process technology has enabled previously unthinkable architectures to be implemented. At the same time, it has forced architects to reevaluate system bottlenecks. With every generation of architecture, complexity is incrementally added to smooth out the kinks— a few more register file ports, a few more entries in the re-order buffers, some more branch prediction bits, slightly larger caches. The bad news is that the burden of verifying and implementing these incremental improvements grows disproportionately with respect to the ultimate performance gain. Design teams are beginning to fold under the weight of their own designs, and thus the wheel of reincarnation turns again— computer architects are rethinking their approach and returning to simpler designs.

Instead of trying to beat the current turn of the wheel of reincarnation, this work will attempt to address the issues of designing for a scenario one or two turns beyond the current turn. This chapter will describe this scenario and some of its basic properties. Chapter 2 will briefly review some ideas, new and old, in the context of this scenario. Chapter 3 presents the core ideas of this thesis, and Chapter 4 concludes with a discussion of the challenges that must be met and the goals that must be accomplished by the terminus of this effort.

1.1 Technology Scenario

This work rests upon the following claims:

- process scaling is leading to clock frequencies f_c with wavelengths $?_c$ significantly shorter than the characteristic dimension D_c of the system
- the ultimate packing density of performance-critical circuitry will be limited not by lithography and minimum wire size, but by dimensional requirements set by a combination of performance requirements, innate electrical parasitics and thermal dissipation requirements.

The first claim is almost true today for off-chip interconnect, and has been true for a while for on-chip interconnect. With high-end processors clocking at 1 GHz, the free-space wavelength is about 300 mm; thus, it is relatavistically impossible to have single-cycle access to any piece of data more than six inches away from the processor. Once the overhead of intermediate circuitry, setup and hold times, and lossy transmission lines are considered, the actual maximum radius of single-cycle access becomes significantly smaller. Thus, as clock speeds continue to increase, it becomes clear that the traversal of any conventionally sized computer system will require more clock cycles.

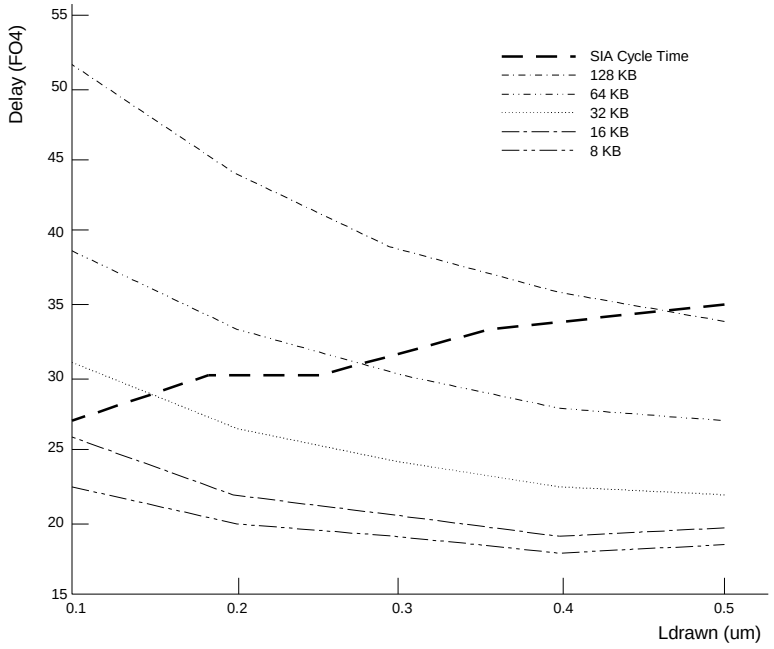

Figure 1-1: Plot of predicted L1 cache delay versus line width versus cache size. [MCF97]

The significance of the second claim is that the reduction in minimum feature size due to scaling will not be ably to buy us the density we require to get around the limitations imposed by the first claim. In other words, as clock speeds escalate, the maximum amount

of memory one can pack within a clock-boundary radius of the processor core is either constant or decreasing as technology progresses, especially in the on-chip realm. Even though transistors are becoming smaller and faster, the optimally buffered wire delay is not quite keeping up. The first-order delay model for optimally buffered wires predicts a linear scaling of wire delay with device scaling. [BAK90] However, recent designs are demonstrating that wire delays are exhibiting worse scaling properties than predicted by the simple models. In particular, the delays of on-chip cache memories have not kept up with the clock rates of microprocessors. One can see from Figure 1-1, Cache Delay vs. Feature Size, that the situation is not getting any better. Table 1-1 summarizes the L1 cache performance of some commercial processors, and gives credence to the predictions of [MCF97]. L1 cache latency is a critical number for processor performance; increased load latencies impacts IPC and requires a more complicated microarchitecture to keep the execution units busy. Despite this, one can see a trend toward multi-cycle L1 caches, driven by the conflicting requirement for a larger L1 cache.

Thus, in the technology scenario for this thesis, increasing processor clock rate may even degrade overall performance. The addition of latency-hiding microarchitectural features to compensate for the higher clock rate grows the processor core area, which again exacerbates the latency issue. For example, the pieces of the hardware required to support out of order execution grows roughly as the square of the reorder depth. Also, the effectiveness of caches diminishes rapidly with increasing cache size.

Processor	Clock Rate	L1 Data Cache Size	L1 Cache Latency
MIPS R3000	33 MHz	64 KB	1
Intel Itanium	~800 MHz	16 KB	2
Alpha 21264	~800 MHz	64 KB	3
AMD Athlon	1000 MHz	64 KB	3
Intel Pentium III	1000 MHz	16 KB	3
Intel Pentium 4	1400 MHz	8 KB	2

Table 1-1: Sampling of L1 cache latencies versus clock speed for various commercial processors. [CHA94] [INTW1] [AMDW1] [CPQW1]

To confound matters, off-chip wiring delays between distributed nodes will grow between $?(\sqrt{N})$ and $?(\sqrt[3]{N})$, even if the topology provides for log(N) worst-case hops. This is because even though transistors are becoming smaller, one cannot pack more nodes into a fixed volume due to the dimension requirements on wires set by parasitics and thermal dissipation requirements. In the limit that each node is already as small as can be, adding more nodes means a growth in real space consumed by the machine.

Given that these technology claims are true, continued performance enhancement in the future will require decentralized processor architectures with distance-aware work-sharing mechanisms.

1.2 The Challenge of Spatially-Aware Decentralized Architectures

1.2.1 Latency and Synchronization

A decentralized computer architecture is one which distributes its processing and memory elements throughout the physical body of the machine. This is in contrast with classic von Neumann architectures that feature a "Central Processor Unit". Most massively parallel architectures can be classified as a kind of decentralized architecture, although the granularity of decentralization varies from PC-based cluster machines to cellular automata machines.

The primary challenge of designing a decentralized architecture lies in dealing with computational situations which require centralized control, or more generally, efficient global connectivity. Events that require global connectivity fall into two related categories: synchronizing events, and memory events. [IAN88] Synchronizing events involve two or more control flows agreeing upon a space and time. Memory events typically involve a single control flow accessing a piece of remote memory. Some parallel architectures implement synchronizing events using special memory events; other architectures provide explicit synchronization resources.

Minimizing the overhead of synchronization and memory reference latency is the challenge of parallel architecture. Interconnect implementations are rapidly approaching the speed of light limited domain of operation; architectures have been proposed that could hit this barrier [DEH93]. Thus, the interesting questions are how to efficiently utilize this network, and how to beat this speed of light limitation. Many architectures address the issue of efficient synchronization and latency hiding. The architecture proposed here contains a blend of old and new ideas tailored toward efficient synchronization and latency reduction through a spatially-aware dynamic object migration mechanism that minimizes the physical distance between points of use.

1.2.2 Reliability

A problem sometimes overlooked or treated as an afterthought by computer architects is the reliability of large, complex systems. The simple fact is that they aren't reliable; components will fail and bit error rates are non-zero. Unfortunately, most architectures are very symmetric, have a rigidly defined topology, and make assumptions about the uniformity of components. The ability to migrate objects around the system easily, as required by the latency management criteria, provides a measure of reliability and fault tolerance. Failing nodes can have objects migrated out of them, and asymmetries in the machine organization can be accounted for by adjusting the object migration and distribution algorithm. A proper implementation would also allow the machine can also be dynamically upgraded or serviced with newer, faster nodes without interruption of service.

1.2.3 Good Single-Threaded Code Performance

Some parallel architectures previously proposed have had problems with good single-threaded code performance. Despite the advantages of parallelism and the arguments set forth in this chapter about the limitations of single-node performance due to the limited packing density of memory around a processor, single-node, single-threaded code performance is still one of the most important factors in determining system performance. [SCO96] This is caused, in part, by the fact that parallel speedup is limited by the time required to execute any sequential portions of the program (Amdahl's Law).

The architecture proposed by this thesis attempts to keep single-threaded code performance high despite the provision of primitives for parallel synchronization and latency management. This goal is met in part by design philosophy. The architecture is based on a simple, five-stage pipelined RISC core; all parallel primitives added to this core must incur minimal overhead when running single-threaded code.

1.2.4 Implementation

The architecture proposed in this work is grounded in the physical world— latency reduction through spatial awareness. As a result, one important aspect of the architecture is its implementation, and the spatial and physical constants that are a result of real-world concerns. The implementer of this architecture is presumed to be working in situation similar to a fabless semiconductor design house. The standard set of tools, processes and macros are available, and a very limited number of full-custom blocks are available as well; the majority of the design will be coded in an HDL such as verilog or VHDL. These implementation assumptions have a non-negligible impact upon the performance targets and physical constants used in the analyses of this thesis.

In order to stay true to this implementation model, throughout this work trial implementations will be run of key architectural components to check assumptions made about their performance and size. Assumptions that fail the reality check will require a re-investigation of the architectural techniques proposed in the thesis. [DALJ98], a retrospective on the J-Machine, provides some commentary upon the importance of implementation in computer architecture research in the section on "Lessons Learned".

2 Background

The genesis of the architecture proposed in this thesis lies in the Dataflow architectures, Decoupled-Access/Execute (DAE) architectures, and Processor-In-Memory (PIM) and Chip Multi-Processor (CMP) architectures. As the basic principles of this machine eschew complexity, superscalar, out of order, and other techniques such as VLIW that require significant circuit-level complexity are mentioned only in passing. In addition, prior works in load balancing and object distribution for large parallel machines are discussed here.

2.1 Dataflow

The architecture proposed in this paper 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 their evaluation context. The length of the instruction run and evaluation context method encapsulated within a token can characterize the spectrum of dataflow architectures. At one end lies the MIT Tagged-Token Dataflow Architecture (TTDA), where each token represents roughly one instruction and its immediate dependencies and results, and token storage is managed implicitly. This evolved into the Monsoon architecture, which has explicit evaluation context management and single-instruction tokens. 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 P-RISC and *T, 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. [ARV93], [NIK89] One could take this one step further and claim that an 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. [LEE94] 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. Because tokens can be formed and dispatched before dependencies are resolved, thousands of tokens are created in the course of even a simple program execution. [ARV93] 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. This is associative structure is not implementable even after twenty years of process scaling.

Another flaw of the early Dataflow machines is that every token represented a high-overhead synchronization event. [IAN88] points out that von Neumann architectures also perform a synchronization event between each instruction, but its 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 architecture proposed by this thesis: 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. The *T architecture focuses primarily on latency hiding through rapid and efficient thread scheduling, starting, and context switching. While this is an important part of the architecture proposed here, it is also very important to reduce latency by providing mechanisms for the efficient profiling of communications patterns and the intelligent migration of data between processor nodes. The machine organization and design reflects this goal of providing a virtual programming interface that efficiently masks key details of the machine's organization without being so opaque as to encourage the programmer to do naughty things that hurt performance. The proposed architecture is also targeted at a conventional programming model similar to Java that can easily leverage the parallelism available in the architecture. Finally, a careful examination of the implementation strategy outlined in [PAP93] reveals a number of important differences (and similarities) between the architecture proposed here and *T. One significant difference is this architecture's use of a queue-based interface between threads, with implicit synchronization through empty/full bits, similar to the scheme used in the J-Machine. [NOA93] *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 this architecture helps cushion network congestion and scheduling hiccoughs.

2.2 Decoupled-Access/Execute

Decoupled-access/execute (DAE) machines hide memory access latencies with architecturally visible queues that couple separate execute and access engines. 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 [SMI82] and later implemented as the Astronautics ZS-1 [SMI87]. [MAN90] characterizes the latency-hiding performance of the ZS-1 in detail, and [MAN91] compares the performance of the ZS-1 to the IBM RS/6000. A comparison of DAE versus superscalar architectures can be found at [FAR93], and a comparison of DAE versus VLIW architectures can be found at [LOV90]. 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 [VEI98]. The architecture proposed in this thesis parallels many of the ideas in [VEI98].

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 is 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. 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 ultimately find themselves drowning in synchronization overheads.

Another important message is that queues are like bypass capacitors in computer architecture. They 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. An area of research for this thesis is the modeling of computation in a queue-rich environment, and exploring the impact of this upon multiprocessor communications and organization.

Unfortunately, DAE machines as a whole are plagued with 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. DAE is very specifically 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 one allows the physical access and execute units to be assigned dynamically to a single virtual control thread. The ability to create 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. The architecture proposed in this thesis allows such virtual DAE machines to be created in a compiler-friendly fashion.

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

PIM architecture has recently become accessible due recent advances in process technology. [TSMW1] [IBMW1] [MOSW1] Very fine-line geometries have made it possible to integrate a sufficient amount of SRAM on-chip to make an interesting processor node. Also, the availability of DRAM embedded on the same die as a processor opens the door to even more interesting possibilities. 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. The advent of extremely high integration process technology has also made it possible to put several processor cores on a single silicon die. A paper that summarizes some of the key arguments for CMP architectures can be found in [OLU96]. Some architectures that have been proposed which take advantage of some combination of embedded memory technology and chip multiprocessor technology include RAW [LEE98], I-RAM [KOZ97], Active Pages [OSK98], Decoupled Access DRAM [VEI98], Terasys [GOK95], 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. [MOSW1] 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 0.13µ process— to entirely eliminate the need for data caches in the processor design. Note that the processor frequency target for this thesis work is on par with compiled 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 nVIDIA series of graphics processors are the canonical speed benchmark used for compiled processors in this work. Private conversations with employees at nVIDIA and observations of the die layout reveal that their processors are designed using a behavioral HDL design flow, and that physical design is done using primarily synthesis tools and timing guided placement. Anecdotal evidence gleaned from websites indicates that nVIDIA achieves 200 to 250 MHz performance at the 0.15µ technology node. 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 [IBMW1] versus 1.9 mm^2 per Mbit for a MoSys macro on a TSMC 0.13µ logic process [MOSW1]).

The architecture proposed in this work 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.

2.4 Latency Reduction

There is a large body of relevant work in the area of data placement and thread migration as latency optimization techniques in parallel machines. Many techniques are specific to a particular class of machine, with the closest relative of the architecture proposed here being NUMA (non-uniform memory access) style machines.

The two broad categories of migration techniques are data migration and computation migration; many techniques combine aspects from both categories. Data migration is the movement of data toward a point of computation to reduce latency; dynamic page-

migration techniques explored in [CHA94] and [NIK00] are examples of this. Simple page migration techniques have been shown to improve performance on NUMA machines by over a factor of two in many cases. Computation migration is the movement of code execution toward synchronization points (other code) or accessed memory. Examples of this include the fine-grained Active Threads explored in [WEI98] and computation migration in [HSI93] and [HSI95], and the coarse-grained dynamic process migration explored in [ROU96]. Techniques to make data placement easier through better programming languages has also been explored [CHA93] [CHA94], and a method for dealing with pointers through thread migrations is outlined in [CRO97].

A general observation of all the works surveyed so far is that the architectural overheads for migration are enormous— some papers quote thread migration times in milliseconds or hundreds of microseconds, often times due to the need to perform asynchronous traps to the kernel. While significant performance gains were demonstrated in the face of such overheads, some papers reported a near net zero gain. [NUT97] [ROU96] Making thread migration, data movement and profiling cheap is a key objective of the architecture proposed here. Another important objective is the integration of migration into the system as a whole, from the language level to the hardware level. Part of the reason why most migration schemes suffered from such high overheads is that they are an afterthought or retrofit upon existing architectures. A cleanly integrated migration scheme will not only reduce overheads, but also make it easy for programmers to leverage migration in their code.

3 The Q-Machine

This thesis proposes a decentralized architecture that is spatially-aware: the Q-Machine. This architecture provides a rich synchronization namespace, a low-overhead synchronization mechanism, a low-latency, scalable interconnect topology, an introspective mechanism for profiling machine behavior and status, and a mechanism for transparently migrating objects around the machine. This architecture is also designed to give good single-threaded code execution performance despite its support for a native multiprocessor environment.

3.1 Programming Model of the Q-Machine

The programmer's-eye view of the Q-Machine is very simple, but slightly novel. A programmer develops code in a sanitary object-oriented environment with garbage collection. The object paradigm implies an intimate pairing of code and data, and this information is made available to the architecture so that it may optimize data access patterns more easily. Figure 3-1 summarizes the machine model presented to the coder.

3.1.1 Method Invocations

Every method invocation is treated as a new thread insofar as the local temporary namespace is re-used. Thus, there is no concept of a stack or stack frames, as there is no concept of a procedure call: each method invocation has a unique context ID and a private set of 128 names with queue semantics. These lightweight threads are probably better described as "continuations", because when they block on a queue becoming empty or full, their state is recorded as a register frame and an instruction pointer. Unlike the dataflow machines, these continuations cannot be passed around during normal operation as a token, as this would incur by far too much communications overhead. Continuations are executed

on the local processor until the garbage collector decides to migrate the continuation to another processor for either load balancing or locality reasons.

3.1.2 Virtual Queue File

The Q-Machine uses queues in lieu of registers in the structure that is conventionally referred to as the "register file". Thus, successive writes to a queue does not clobber the preceding writes' data (although at the assembly level, this option is provided for): rather, the values queue up until successive reads from the queue consumes these values. Reads from an empty queue cause the machine to stall or block; writes to a full queue also block. As mentioned previously, an option is provided in the instruction opcodes that cause values to be copied or clobbered, so that a queue can be used as a normal register if the compiler so desires. As a final twist, the heads and tails of queues may be remapped to different contexts, thus allowing for a synchronized mechanism for inter-thread communication. This remapping happens via the network interface to allow for the possibility that the source and destination contexts do not exist on the same physical node. Arguments are passed between threads through remapped queues. Knowledge about remapped queues are stored as pointers in memory, so that a reference counting garbage collection mechanism can be used to efficiently migrate continuations about the machine through the use of forwarding pointers.

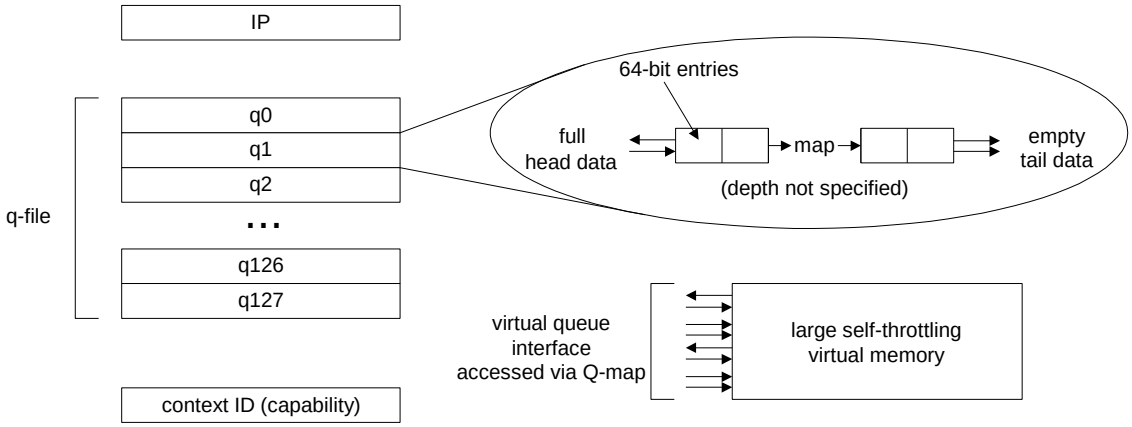

Figure 3-1: Programmer's model of the Q-Machine

3.1.3 Memory Subsystem and Network Interfaces

As far as the programmer is concerned, memory is also accessed via queue structures. A pair of queues is allocated for a load or a store, representing the address and data path for these operations. Addresses can be speculatively generated and queued before the first piece of data becomes available. Multiple memory queue maps can be allocated for a thread.

Tables of network communication statistics are accumulated in a set of registers that are not visible to the applications programmer. The accumulation of statistics is accomplished by a fiduciary processor, so that there is no performance penalty associated with this operation. The operating system accesses these tables during garbage collection, and as part of the GC algorithm, objects may be moved to optimize access latency between objects.

Finally, there is no coherent global shared memory in this machine, as is the case on ccNUMA machines. Instead, a large shared address space is presented to the user, where the physical node that owns a piece of data is hard-coded into the address. Thus, method threads running on an object must be located on the same physical node as the storage for that object. This discipline is maintained in part by the way context IDs are represented in the Q-Machine's implementation.

Each processor node has a large virtual memory space. The memory hierarchy is structured so that it is self-throttling; allocating large amounts of memory on a single processor will dramatically increase the access latency to memory, so that the allocation process itself slows down. This gives the GC a chance to run and move objects out of that node before the physical memory is exhausted.

3.1.4 Opcodes, Addresses and Data Types on the Q-Machine

The format of instructions, addresses and data types on the Q-Machine is given in Figure 3-2. A table for an initial ISA for the Q-Machine is given in Table 3-1.

Branch instructions encode in them a 12-bit branch history field, which is updated by the instruction fetch unit based on the direction that the branch took after the arguments are evaluated. The history field is written back into the instruction cache, but no effort is used to

ensure global coherence of this field between processors. The compiler may specify the initial value of this field if it wishes to be clever, but by default, it is either initialed to all "branch taken" or "branch not taken". The branch history field allows the instruction prefetcher to make better decisions without actually consuming any processor real-estate for a branch prediction table. This technique gives an effective branch prediction table as large as the instruction memory. It can also give persistent branch predictions if the values eventually find their way to the original binary code written to disk.

Jump opcodes encode in them a 28 bit destination storage field. This field, like the branch history bits in the branch opcode, is updated every time the jump is taken, written to the instruction cache, but no effort is consumed to ensure global consistency. The destination field allows the instruction prefetcher to speculatively fetch the next instruction after a jump; some check and rollback mechanism is required in the case that the actual jump destination is different, as would be the case in a dynamic dispatch table.

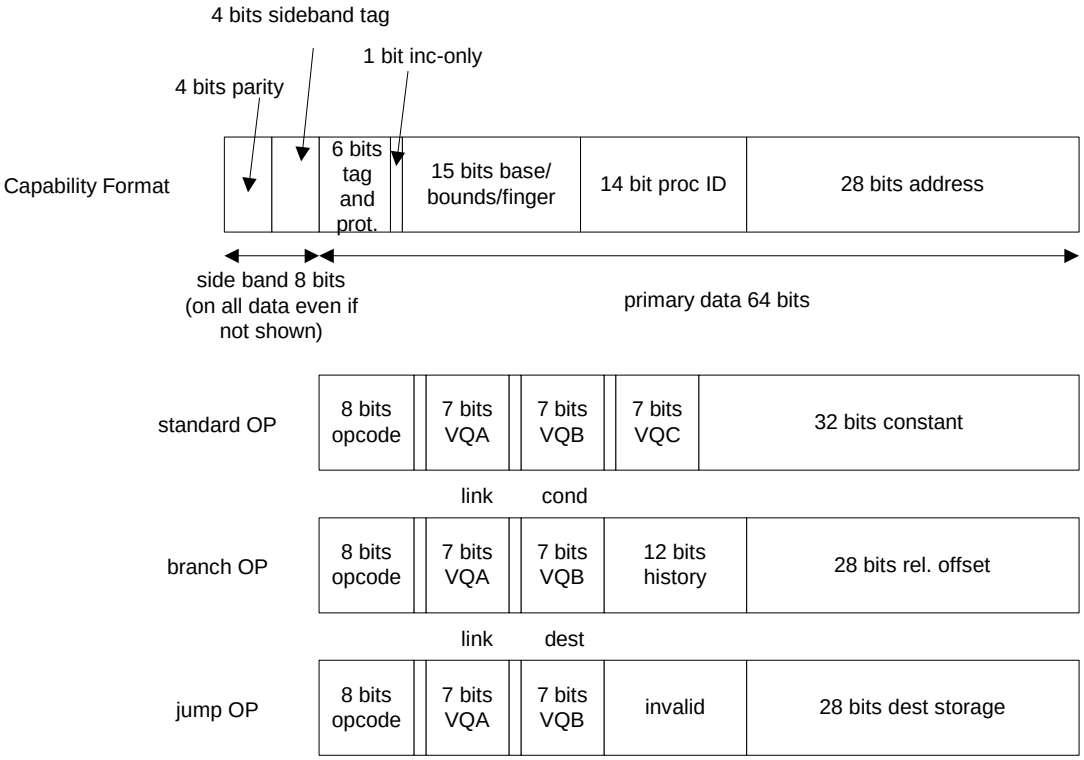

Figure 3-2: Capability and basic opcode formats. Memories are word-addressed, where a word is 32 bits.

In order to accelerate GC and to help make continuation storage in memory more efficient, addresses are represented using a capability format. [BRO00] This capability format allows methods to extract a conservatively correct object base and bounds given any valid pointer. Notice how the capability format explicitly codes the processor ID in the address field. This is used to accelerate object migration and backing storage allocation, as described later in this proposal. Other uses for the capability format, including sparse object allocation across multiple processors, are outlined in [GRO01].

One will also note that while the addressable space of the machine is limited to 16 TB, an individual node can only address up to 1 GByte of virtual memory. This limitation is considered acceptable because one of the critical assumptions of this thesis is that physical distance to memory combined with the packing density of memory cells places a lower bound on the access latency to memory. Any programmer interested in optimal performance should not be allocating 1 GByte monolithic objects on a single node; rather, a distributed approach should be taken to allocating very large objects to enhance performance and take advantage of the fine-grained multiprocessor nature of the Q-Machine.

```
ADD qa, qb, qc              - qc <- qa+qb
SUB qa, qb, qc              - qc <- qa-qb
MUL qa, qb, qc              - qc <- qa*qb
DIV qa, qb, qc              - qc <- qa/qb
ADDC qa, n, qc              - qc <- qa+n
SUBC qa, n, qc              - qc <- qa-n
MULC qa, n, qc              - qc <- qa*n
DIVC qa, n, qc              - qc <- qa/n
AND qa, qb, qc              - qc <- qa & qb bitwise
OR qa, qb, qc               - qc <- qa | qb bitwise
XOR qa, qb, qc              - qc <- qa ^ qb bitwise
SHL qa, qb, qc              - qc <- qa << qb bitwise
SHR qa, qb, qc              - qc <- qa >>> qb bitwise, 0 fill
SRA qa, qb, qc              - qc <- qa >> qb bitwise, sign extend
ANDC qa, n, qc              - qc <- qa & n bitwise
ORC qa, n, qc               - qc <- qa | n bitwise
XORC qa, n, qc              - qc <- qa ^ n bitwise
SHLC qa, n, qc              - qc <- qa << n bitwise
SHRC qa, n, qc              - qc <- qa >>> n bitwise, 0 fill
SRAC qa, n, qc              - qc <- qa >> n bitwise, sign extend
SEQ qa, qb, qc              - qc <- qa == qb ? 1 : 0
SNE qa, qb, qc              - qc <- qa != qb ? 1 : 0
SLT qa, qb, qc              - qc <- qa < qb ? 1 : 0
SGT qa, qb, qc              - qc <- qa > qb ? 1 : 0
SLE qa, qb, qc              - qc <- qa <= qb ? 1 : 0
SGE qa, qb, qc              - qc <- qa >= qb ? 1 : 0
SEQC qa, n, qc              - qc <- qa == n ? 1 : 0
SNEC qa, n, qc              - qc <- qa != n ? 1 : 0
SLTC qa, n, qc              - qc <- qa < n ? 1 : 0
SGTC qa, n, qc              - qc <- qa > n ? 1 : 0
SLEC qa, n, qc              - qc <- qa <= n ? 1 : 0
SGEC qa, n, qc              - qc <- qa >= n ? 1 : 0
FADD qa, qb, qc             - qc <- qa+qb (float)
```

```
FSUB qa, qb, qc             - qc <- qa-qb (float)
FMUL qa, qb, qc             - qc <- qa*qb (float)
FDIV qa, qb, qc             - qc <- qa/qb (float)
FADDC qa, n, qc             - qc <- qa+n (float)
FSUBC qa, n, qc             - qc <- qa-n (float)
FMULC qa, n, qc             - qc <- qa*n (float)
FDIVC qa, n, qc             - qc <- qa/n (float)
FCPYS qa, qb, qc            - qc <- qa[63] || qb[62:0]
FCPYSEX qa, qb, qc          - qc <- qa[63:52] || qb[51:0]
FCPYSN qa, qb, qc           - qc <- NOT(qa[63]) || qb[62:0]
BR label                    - ip <- index(label)
BRL label, qc               - qc <- ip, ip <- index(label)
BRZ qa, label               - qa == 0, ip <- index(label)
BRNZ qa, label              - qa != 0, ip <- index(label)
JMP qa                      - ip <- qa
MOVE qa, qc                 - qc <- qa
MOVEC n, qc                 - qc <- n, where n is long or float
FLUSHQ qc                   - set qc empty
MML qi, qj                  - map 'qi to load addr, 'qj to data return
MMS qi, qj                  - map 'qi to store addr, 'qj to data
PROCID qc                   - qc <- thread.context_id
FORK label, qc              - new thread starting at label,
                              qc <- new thread.context_id
MAPQ qi, qj, qa             - 'qi mapped to 'qj in context qa
MAPQI qi, qb, qc            - 'qi mapped to qb in context qc
UNMAPQ qi                   - 'qi unmapped in current context
CONSUME qa                  - read from qa, discard
EEQ qc                      - nop conditional on empty qc
HALT arg1, arg2             - kill current thread

Above, 'qi means the literal queue, not the value stored in the queue.
```

Table 3-1: Rough ISA for the Q-Machine

3.2 Implementation of the Q-Machine

The Q-Machine consists of two basic components: nodes and network.

3.2.1 Nodes

The basic structure of a Q-Machine node is illustrated in Figure 3-3. A node consists of an execution core, a network interface, a thread scheduler/performance monitor, and a virtual memory subsystem.

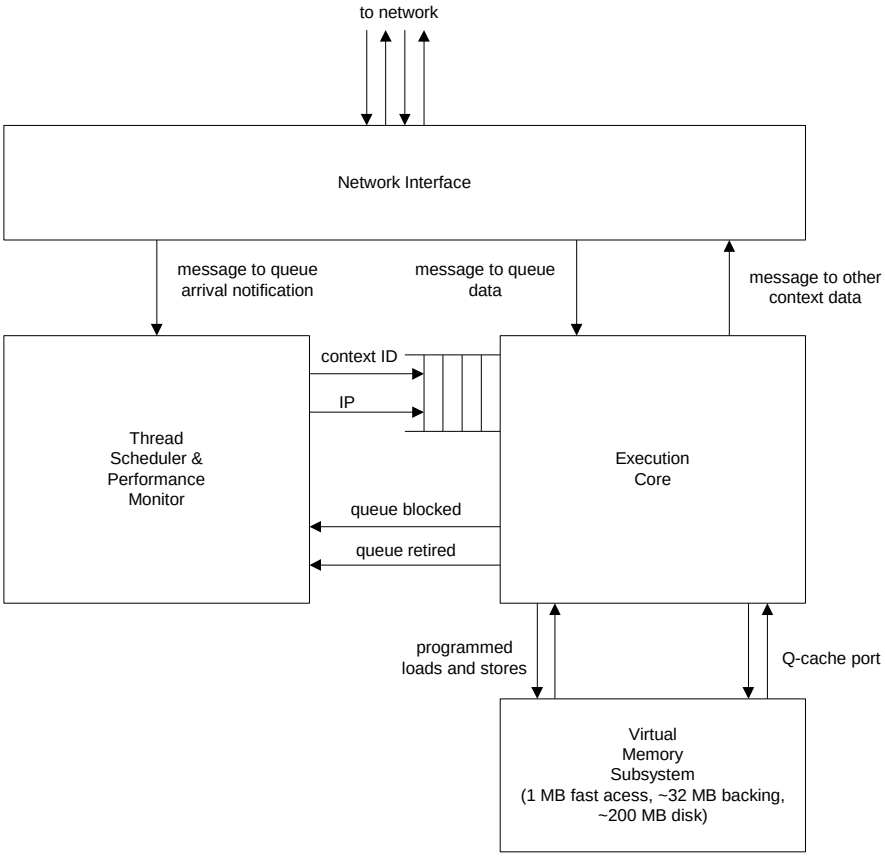

Figure 3-3: Basic Q-Machine node structure. The dimensions of the memory subsystem are approximate.

3.2.1.1 Execution Core

The organization of the execution core is illustrated in Figure 3-4. The execution core consists of the virtual queue file (VQF), a MAP box, an ALU/MEM box, and connections to the network interface and thread scheduler. The diagram does not show pipeline stages for clarity; however, the entire execution core pipeline is kept very short and simple so that stalls and restarts are fast. A very simple scheduler built into the core that observes the incoming work queue and the state of the machine enables limited out-of-order execution between different thread contexts in the case of stalls and blocks.

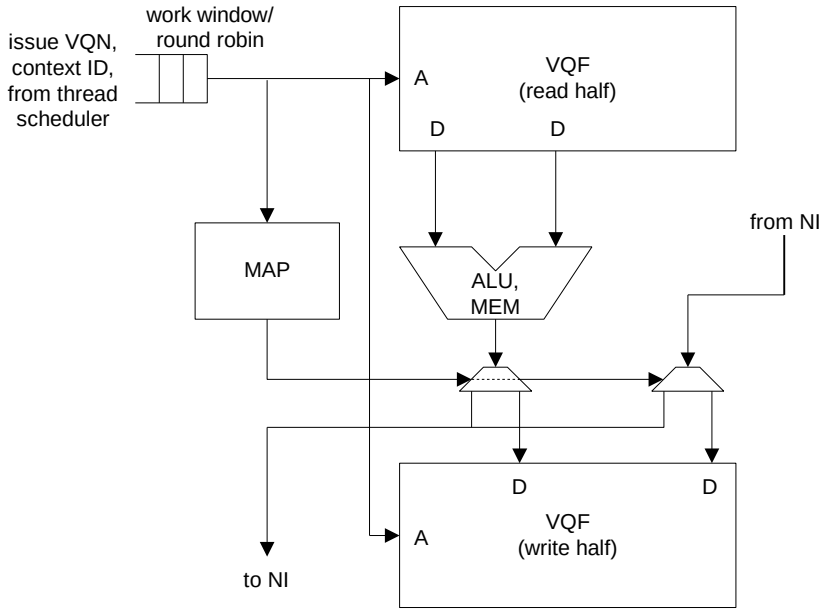

Figure 3-4: Execution core overview.

The virtual queue file (VQF) implements the following functionality:

- given a read command, `read <context ID, Q #>`, return the bottom piece of data at that queue or a block if there is no data
- given a write command, `write <context ID, Q #, data>`, write the given data to the top of the specified queue, and block if that queue is full

The astute reader will notice by this point that the VQF as described is not feasible for direct implementation. The fact that a virtually unlimited number of thread contexts are available at the drop of a hat to the programmer should have been cause enough for alarm. The truth is that these contexts don't actually exist in exactly the same way that the majority of the memory pages in a virtual memory system don't exist. The context ID for a thread continuation corresponds directly to the virtual address used as backing storage for the context's active queue state and local variables. Thus, the VQF itself has a memory hierarchy. At the heart of the VQF is the physical queue file (PQF), which directly implements an architecturally unspecified number of queues. 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 Q-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 GC'd.

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.

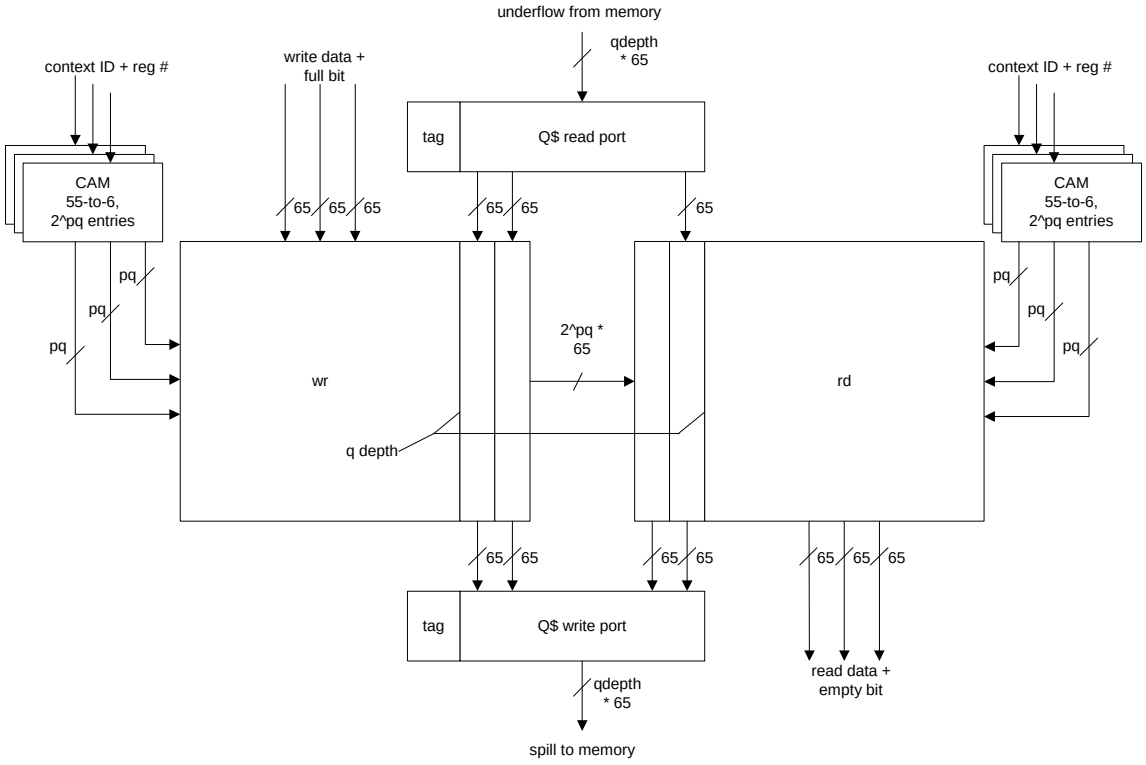

Figure 3-5: A 3-write, 3-read port VQF implementation. pq = $\log_2$(# physical registers). Q-cache details omitted for clarity.

The physical queue file actually does not take up significantly more space 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 3-6 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 0.18u process guide [TSMW1]. The wiring requirements for the unit cell of the PQF would consume 4851 $?^2$ alone, using minimum-pitch M5/M6 wires. For comparison, the area of a 6-T SRAM cell in the TSMC 0.18u process is 574 $?^2$, 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.

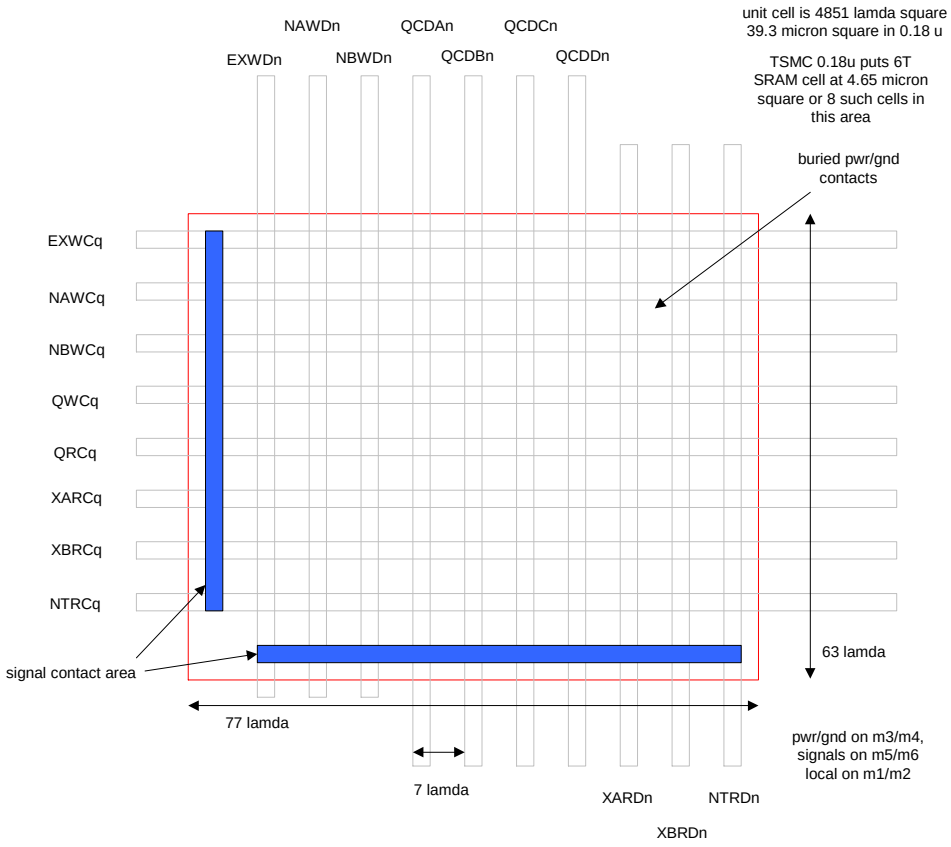

Figure 3-6: PQF unit cell.

Hence, a PQF implementation which has relatively shallow queues (4 to 8-deep) can 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 similar idea to the VQF implementation outlined here is the Named-State Register File (NSRF). [NUT91], [NUT95] 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. [NUT95]

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 Figure 3-2 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.

The ALU/MEM subsystem is relatively straightforward. The ALU implements all the standard execution core functions— integer and floating point arithmetic, compares, and logicals. It is pipelined to a depth of two or three stages. The MEM subsystem contains a fast path to the local on-die memory. No caches are required in this path; the on-die memory is designed to return hits within two to three processor cycles, plus a cycle for the TLB operation. Memories on a likely target process, such as the TSMC 0.13u process, already exist which achieve this performance goal, and they can be purchased as hard-cores from Mosys. [MOSW1]

Let us now take a moment to consider the performance of a single node on single-threaded code. In the case that a single thread is running on the machine, the PQF, if it has at least 128 queues, is capable of holding an entire context in-core. Thus, no stalls are incurred when executing from a single context. Also, the MAP function is able to complete its determination that no mappings occur within the latency of the arithmetic pipeline, so no stalls are incurred as a result of that, either. Finally, the memory subsystem of the Q-Machine is fast, or at least on par with conventional uniprocessors. The local ~ 1MByte of data is accessible within a couple of clock cycles (on par with the L1 cache latencies of conventional processors on the market today), and access latency to the virtual memory subsystem is on par with conventional processors. Thus, the entire critical path of the Q-Machine node resembles, to a good approximation, that of a conventional RISC machine when executing single-threaded code. This is in-line with the goal of not sacrificing single-threaded code performance on the Q-Machine.

3.2.1.2 Thread Scheduling

The basic unit of execution on the Q-Machine is called a thread. Threads are very lightweight on the Q-Machine, and are similar in nature to continuations or microthreads in other architectures, such as the Dataflow machines. [PAP93] The Q-Machine architecture is similar to the hybrid dataflow architectures in that thread state is not passed around as a part of tokens on the network; rather, a message may be passed to activate or invoke a thread, but the actual migration of thread data is considered to be a high-cost operation that is managed by the garbage collector. A subtle difference between the dataflow machines and the Q-Machine is that the duration of the run-time for a thread varies dramatically depending upon the demands of the user program. A thread of execution may run for a long period of time, as is the case when the working set for the local processor exhibits good locality and poor parallelism. Other threads may execute for just a few instructions, as would be the case on a sparse-matrix operation, or on a memory reference thread.

Because procedure calls are treated like threads, the number of threads on a single processor can grow quite rapidly. In order to manage this potential explosion of threads, the thread scheduler imposes a hierarchical structure on the thread state of the machine. This structure is outlined in Figure 3-7.

There are two general classes of threads on a Q-Machine node: active and retired. Active threads are considered for execution on each round of scheduling; retired threads are swapped out until some condition (typically a queue blockage) is resolved. Each object on the machine is given a light-weight server thread that always remains in the active set of scheduled threads. Beneath each object is a set of threads grouped by method invocation; these threads are referred to as a *method group*. A method group is scheduled essentially depth-first, under the premise that each method is implemented in a single-threaded frame of mind. Method groups may not be executed in strictly depth-first order, because queues used to link arguments between procedure calls within the group may block because they are full, and the predecessor to the bottom-most active thread may have to scheduled in again to drain the last few arguments waiting to be passed. This method of dividing threads into groups is similar to the H- and V-threads scheme used in the M-Machine. [FIL95] Similarly, a method for scheduling threads (continuations) based on empty/full bits of work queues is used in the J-Machine [NOA93].

Thus, the active set of threads consists of object servers and the bottom-most procedure call of each method group. These threads in the active set are scheduled in some fashion that guarantees fairness without starvation. It is a topic of research to discover the optimal scheduling algorithm given the limited set of computational resources available in the Q-Machine scheduler implementation.

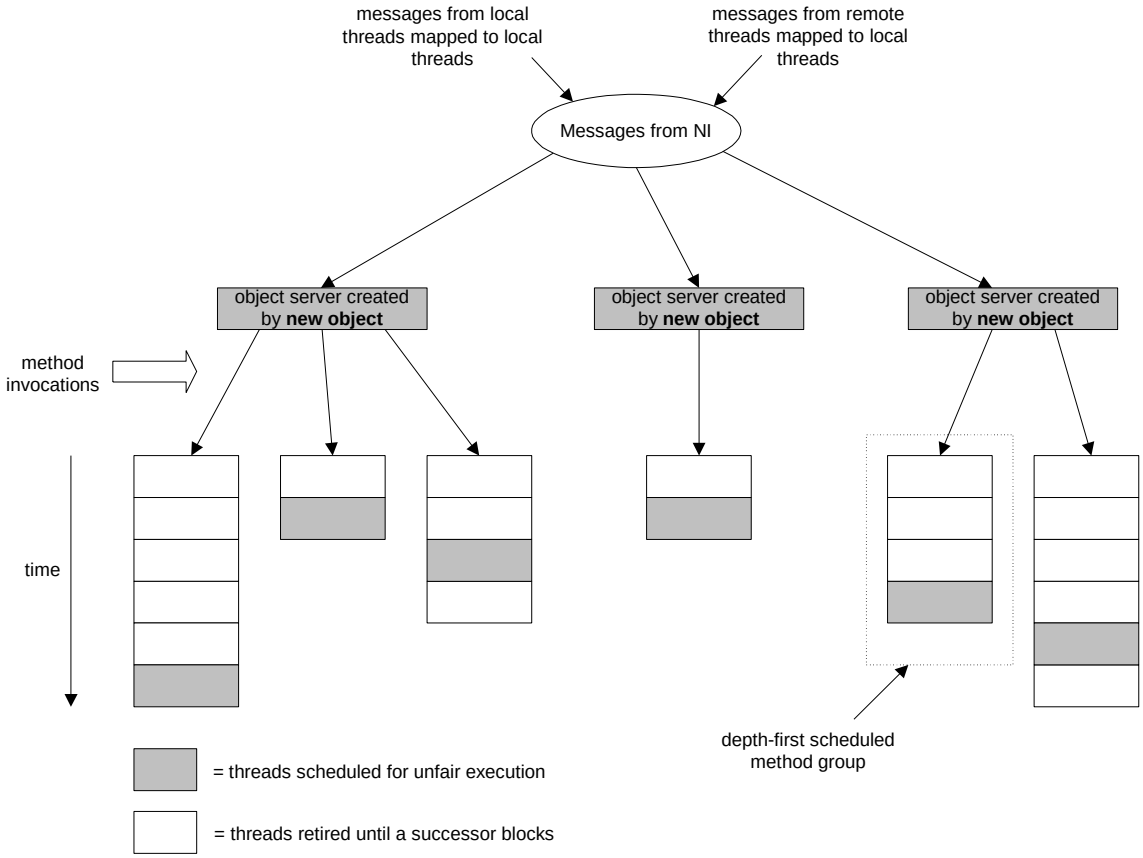

Figure 3-7: Basic thread taxonomy and scheduling chart.

The thread scheduler of the Q-Machine is implemented as a conventional RISC microprocessor with some dedicated hardware resources to facilitate the task of scheduling. Note that the Tensilica [TENW1] configurable microprocessor core, for example, can be implemented in as little as 0.7 mm^2 in a 0.18u process with a typical performance of 320 MHz. The actual area would be about 50% larger because an instruction cache would be required to sustain high performance, but a data cache is not required because the core would be fed by a dedicated MoSys memory macro that can run at processor speeds. [MOSW1] In order to accelerate the task of scheduling threads, application specific hardware is necessary. The total response latency from an incoming network request to a thread being scheduled into the Execution Core's work queue should be a few cycles at most under light loads. Depending upon the area restrictions of the machine, this special hardware will be implemented as a combination of reconfigurable hardware and auspiciously chosen hard-cores. [CUL91] gives insight into an architecture that has addressed similar thread scheduling issues and tradeoffs. [NIK92] describes the *T architecture which has a similar

single-node organization with a separate synchronization coprocessor to handle thread scheduling, and a RISC microprocessor to handle straight-line code execution.

The ability of the thread scheduler to handle its workload will have a strong impact upon the multi-threaded performance of the Q-Machine. It will also have an impact upon the structure of the user's code. A pathological example would be if a user allocated a large array of objects based on the **`Integer`** class that implement a specific computational method, and simply issued a large number of messages to invoke methods on these objects for a massively parallel computation. Each integer would have a thread assigned to it; even if the threads are lightweight, the synchronization and storage overhead is likely to outweigh the benefits of this style of computation (which is, incidentally, very similar to the original Dataflow style of parallelism [ARV90]). The exact breakpoint of how fine of a grain users can break their parallel code into is determined largely by the amount of overhead incurred by the thread scheduler and thread storage in the Q-Machine implementation. One key benefit of the Q-Machine architecture, however, is that distributing the objects over more processors can mitigate this overhead. Another key benefit is that the user has the ability to choose, through coding discipline, a level of granularity for their parallel tasks that spans a fairly broad spectrum.

3.2.1.3 Performance Monitor

An important aspect of the Q-Machine from the systems standpoint is the on-line performance monitor. This is a function likely to be integrated into the thread scheduler unit, but important enough to be mentioned in a section on its own. The performance monitor plays an important role in the programming and debug of the machine, in the run-time environment of the machine, and in the reliability and fault tolerant aspects of the machine.

A key challenge faced by programmers of large parallel systems is performance profiling and the debugging of hairy, multithreaded code. Simply wading through core dumps and attaching **`gdb`** processes to various threads is a tedious and time-consuming process. Similarly, inserting system calls to time functions or standard-output functions such as **`printf`** often incurs locking and kernel overhead that significantly alter the performance of

a parallel program. One of the design goals of the on-line performance monitor is to be able to collect the right kind of data without any overhead to the running program so that programmers can easily discover bugs and tune programs for optimal performance.

The run-time environment for the Q-Machine will also leverage the information collected by the performance monitor to optimize data placement and communications patterns in the machine. The garbage collection process polls the performance monitor for hot-spots and attempts to migrate data and threads to alleviate such hot-spotting. This kind of dynamic data migration and placement has been shown to give significant performance benefits in even coarsely parallel, non-speed of light limited cluster machines such as the Stanford DASH. [CHA94] I predict that dynamic data and thread migration will be crucial in future machines which are heavily impacted by the locality of data due to latency limitations placed by the speed of light.

The performance monitor can also be used to monitor the health of a machine. It can collect information about bit error rates on network interfaces and parity errors in memory cells. It can also be connected to other key system health indicators, such as core voltage and temperature levels or SMART (Self-Monitoring Analysis and Reporting Technology) information from hard drives. If a node seems to be exhibiting signs of failure, a flag can be raised to the garbage collector and data can be migrated out of the node so that it can be powered down and replaced by field service.

3.2.1.4 Virtual Memory Subsystem

The memory hierarchy of a Q-Machine node is illustrated in Figure 3-8. Each node has the capability to address up to 1 GByte of memory; however, users are assumed to seldom take advantage of this large address space. Allocating 1 GByte of objects on a single node violates a fundamental premise of the Q-Machine: optimal performance demands that data and processor be intimately located. The allocation of large amounts of data on a single node should be avoided for good performance scalability. At the same time, forcing users to operate within a strict memory budget hampers the ease of use of the machine. Thus, each node is designed with a fast, small memory co-located with the Execution Core, but has

provisions for an extensive virtual memory hierarchy to give coders the headroom they require to get the job done when performance is not at a premium.

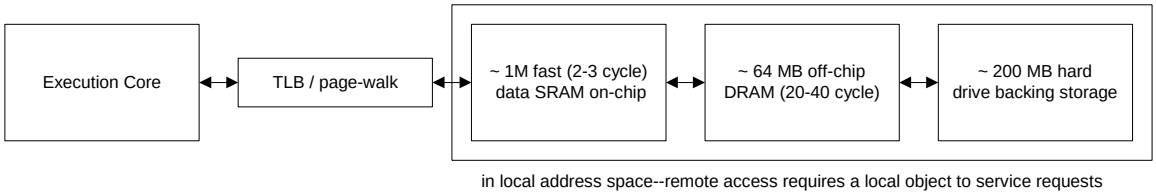

Figure 3-8: Virtual Memory hierarchy of a single Q-Machine node.

A relatively large (64 MB) secondary memory is provided off-chip, backed by an even larger non-volatile storage unit. This hierarchy of degrading performance memory is necessary to prevent allocation run-away on a single node. A careless programmer could then be saved by the garbage collector, which will make efforts to migrate objects out of the thrashing node before its virtual memory is exhausted. At the same time, a performance-conscious programmer can balance object allocation across the machine and get optimal performance from their program under a broader range of conditions.

From the programmer's standpoint, memory is accessed via load and store queues. However, in reality, the far-side of the queues are processes that run on the execution core. The advantage of the queue-based memory access paradigm is that programmers can leverage control and address generation decoupling to hide latency. In addition, programmers can make the memory-side handlers more intelligent and embed operations such as pointer chases and table lookups on the memory-side of the queue.

3.2.1.5 Network Interface and Machine Network

The Q-Machine nodes are embedded in a fast, low latency, fault tolerant network based on the work done by the MIT METRO project. Because of the extensive prior research devoted to this topic, the treatment of he network interface and the network is brief, and interested readers are referred to the original papers on this topic. [DEH93] [Minsky's thesis, leiserson's paper on fat trees]

The network topology used by the Q-Machine is a hybrid multipath fat-tree/multipath multibutterfly topology network. It is a circuit-switched network with self-routing packets

and wormhole routing. The design of the packets and network are optimized for low-latency through the network routers; a single bit or pair of bits sent at the head of the packets determines the state of the router. The router design is kept simple by making the routing protocol source-responsible; blocked paths and errors are reported as dropped message, and it is up to the NI to resend the message. Fault tolerance and congestion avoidance are designed into the network by proper choice of topology; the network has a dilation (redundancy) factor of two, and the wiring between nodes is random with a maximal fan-out property, so that no individual component failure will leave a destination node unreachable.

All clocks on the Q-Machine originate from a single frequency source. Mesochronous clocking with wave pipelining is used to make the sources and destinations skew-insensitive. The advantage of mesochronous clocking is that no time is consumed in router stages trying to re-synchronize data to the local clock domain. Other designs, such as the CrayLink network used in the SGI Origin 2000 series, devote a large portion of the router time to synchronization for metastability resistance. [GAL]

Because the design of the routers is kept so simple, the dominant latency of the interconnect is wire delay. It is an important assumption of this thesis work that the dominant delay of the system be the wire delays. Estimations based on scaling previous and current implementations of the METRO network indicate that this is a solid assumption.

One downside of making the routers simple is that more complexity is required in the Q-Machine nodes to keep track of open network connections. The implementation of this hardware is a topic of research, but the current plan is to devote a dedicated RISC processor to handling messages along with some reconfigurable hardware, or to integrate this functionality into the Thread Scheduler, which also has a dedicated RISC processor with some dedicated hardware to handle its tasks. Regardless, the network interface and network implementation details are not a major area of focus for this work because of the extensive body of prior work in this area.

Some improvements on the METRO network explored within the context of this research are idempotent messaging and message priorities for deadlock avoidance. A three-phase

protocol is used with temporary message state on both sides of the network to guarantee message idempotence. Interested readers in these improvements are referred to Bobby Woods-Corwin's Masters of Engineering thesis work-in-progress.

4 Directions

The architecture described in the previous section addresses many issues, but it is far from complete. An important part of this research effort is to flesh out the architecture and discover where the bottlenecks are. The metric for success is ultimately measured in terms of how much wall-clock programmer time is required to produce results— from coding to debugging to the actual run-time of the program. While this metric is very difficult to assess due to the variability of human factors, keeping this end-goal in mind gives this research a very systems-oriented approach, for the perfection of any single piece of the machine will not guarantee good results. On the other hand, careful attention to making sure that the nominal case exhibits robustly good performance will have a higher chance of yielding a successful architecture. Simply put, peak performance numbers are bogus, and the important number to optimize is the average performance, even if it hurts the peak performance number. In addition, the worst-case number is not important as long as the worst-case is never in the critical path, and as long as the worst-case happens infrequently. Thus, a central goal of this thesis is validating the system concept behind the Q-Machine through implementation.

4.1 Implementation

There is no substitute for building an architecture. Implementation tests the assumptions that the architecture rests upon, and exposes uncountable oversights that can lead to serious bottlenecks or implementation trade-offs that alter the nature of the architecture. At the same time, it is too easy to become obsessed with tweaking and optimizing small parts of the machine, and miss the big picture. Thus, no chips will be fabricated: rather, this thesis hopes

to answer the question of if it is worth the effort of fabricating chips for this architecture. Hence, the implementation side of this thesis will start with behavioral HDL simulations implemented in FPGAs and RISC emulations cores, and extend as far as estimated die floorplans and trial layouts of critical machine components. In parallel with the hardware development efforts, very high-level simulations of the data migration and object-oriented run-time mechanisms will have to be run to collect crucial information about the expected utilization of key machine components, such as the network, thread scheduler, and on-chip memory.

4.2 Analysis

The implementation effort is in vain without an analysis of the shortcomings and strengths of the Q-Machine. Some of the important questions that I hope to answer and issues I hope to address through the implementation effort include:

- effectiveness of object migration in the Q-Machine
- object migration and garbage collection strategies, and how they are different from previously proposed strategies
- programming languages and paradigms for the Q-Machine
- advantages of the Q-Machine mechanisms over more conventional, non-queue based mechanisms without custom support for object migration
- analysis of latency and area overhead of the synchronization primitives, i.e., Thread Scheduler unit and Network Interface unit
- efficient thread scheduling and thread representation on the Q-Machine
- analysis of potential areas of inefficiency within the Q-Machine architecture, and proposals for their resolution

4.3 Conclusion

The Q-Machine architecture outlined in section 3 is actually a significant portion of the final thesis work. The proposal for research from this point is to implement and analyze the system concept of the proposed Q-Machine. The research will begin with some high-level simulations to extract object interaction and locality properties in Java. This will form a baseline set of assumptions that can be applied to create a machine model. In parallel with that, basic elements of the Q-Machine hardware will be marshaled— the network (from

[DEH93]) and simulation vehicles for the processor nodes will be constructed from FPGAs and conventional RISC processors. From here, the set of assumptions and machine models derived from the high level simulations will be applied to create a detailed specification of implementation requirements for the Q-Machine architecture. These requirements will then be run through trial implementations to see if they are reasonable. Finally, the pieces of the machine that have been implemented will integrated and some simple benchmarks will be executed on the architecture that can validate the models and assumptions generated by the original high-level simulations. The level of detail in the implementation is geared at getting significant results that can clearly validate the Q-Machine system concepts while keeping on track for a graduation target of June 2002.

5 References

[DEH93] DeHon, Andre. "Robust, High-Speed Network Design for Large-Scale Multiprocessing". AI Technical Report Number 1445. September 1993. MIT Artificial Intelligence Laboratory, 200 Technology Square, Cambridge, MA, 02139.

[IAN88] Iannucci, Robert. "Toward a Dataflow/von Neumann Hybrid Architecture". Proceedings of the 15th Annual International Symposium on Computer Architecture. May 30-June 2, 1988. Honolulu, Hawaii.

[SCO96] Scott, Steven L. "Synchronization and Communication in the T3E Multiprocessor." Proceedings of ASPLOS VII, Massachusetts, 1996. ACM, 1996.

[ARV93] Arvind. Brobst, Stephen. "The Evolution of Dataflow Architectures: from Static Dataflow to P-RISC". International Journal of High Speed Computing. Vol. 5, No. 2. World Scientific Publishing Company, 1993.

[NIK89] Nikhil, Rishiyur S. Arvind. "Can Dataflow Subsume von Neumann Computing?" Proceedings of the 16th Annual International Symposium on Computer Architecture, May 1989, Jerusalem, Isreal.

[LEE94] Lee, Ben. Hurson, A.R. "Dataflow Architectures and Multithreading." IEEE Computer. August 1994.

[SMI82] Smith, James E. "Decoupled Access/Execute Computer Architectures". Proceedings of the 9th Annual International Symposium on Computer Architecture. April 1982, Austin, Texas.

[SMI87] Smith, J. E. Dermer, G. E. Vanderwarn, B.D. Klinger, S. D. Rozewski, C. M. Folwler, D. L. Scidmore, K. R. Laudon, J. P. "The ZS-1 Central Processor". Second Internatinoal Conference on Architectural Support for Programming Languages and Operating Systems. October 1987. pp. 199-204.

[MAN90] Mangione-Smith, William. Abraham, Santosh G. Davidson, Edward S. "The Effects of Memory Latency and Fine-Grained Parallelism on Astronautics ZS-1 Performance." Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences, 1990. IEEE, 1990. Pp. 288 –296, Vol. 1.

[FAR93] Farrens, Matthew K. Ng, Pius. Nico, Phil. "A Comparison of Superscalar and Decoupled Access/Execute Architectures". Proceedings of the 26th Annual International Symposium on Microarchitecture, 1993. IEEE, 1993. Pp. 100 –103.

[LOV90] Love, Carl E. Jordan, Harry F. "An Investigation of Static Versus Dynamic Scheduling". Proceedings of the 17th Annual International Symposium on Computer Architecture, 1990. IEEE, 1990 Pp. 192 –201.

[MAN91] Mangione-Smith, William. Abraham, Santosh G. Davidson, Edward S. "Architectural vs. Delivered Performance of the IBM RS/6000 and the Astronautics ZS-1". Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences, 1991. IEEE, 1991. Pp. 397 -408 vol.1.

[WUL92] Wulf, William A. "Evaluation of the WM Architecture". Proceedings of the 19th Annual International Symposium on Computer Architecture, 1992. ACM. Pp. 382 – 390.

[VEI98] Veidenbaum, Alexander V. Gallivan, K. A. "Decoupled Access DRAM Architecture". Innovative Architecture for Future Generation High-Performance Processors and Systems. IEEE, 1997, 1998. Pp. 94-103.

[OLU96] Olukotun, Kunle. Nayfeh, Basem A. Hammond, Lance. Wilson, Ken. Chang, Kunyung. "The Case for a Single-Chip Multiprocessor." Proceedings of ASPLOS-VII, Cambridge, MA. ACM, 1996.

[KOZ97] Kozyrakis, C. Perissakis, S. Patterson, D. Andreson, T. Asanovic, K. Cardwell, N. Fromm, R. Golbus, J. Gribstad, B. Keeton, K. Thomas, R. Treuhaft, N. Yelick, K. "Scalable Processors in the Billion-Transistor Era: IRAM." IEEE Computer. September 1997. Pp. 75 – 78.

[OSK98] Oksin, M. Chong, F.T. Sherwood, T. "Active Pages: A Computational Model for Intelligent Memory." The 25th Annual Symposium on Computer Architecture, 1998. IEEE, 1998. Pp. 192-203.

[MAR00] Margolus, Norman. "An Embedded DRAM Architecture for Large-Scale Spatial-Lattice Computations." The 27th Annual International Symposium on Computer Architecture, 2000. IEEE, 2000. Pp. 149-160.

[GOK95] Gokhale, M. Holmes, B. Iobst, K. "Processing in Memory: The Terasys Massively Parallel PIM Array." IEEE Computer. Vol. 28, Issue 4. April, 1994. Pp. 23-31.

[FIL95] Fillo, Marco. Keckler, Stephen W. Dally, William J. Carter, Nicholas P. Chang, Andrew. Gurevich, Yevgeny. Lee, Whay S. "The M-Machine Multicomputer." Proceedings of the 28th Annual International Symposium on Microarchitecture. IEEE, 1995. Pp. 146-156.

[LEE98] Lee, Walter. Barua, Rajeev. Frank, Matthew. Srikrishna, Devabhaktuni. Babb, Jonathan. Sarkar, Vivek. Amarasinghe, Saman. "Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine." Proceedings of ASPLOS-VIII, California, USA. ACM, 1998.

[BRO00] Brown, Jeremy. Grossman, J.P. Huang, Andrew. Knight, Tom. "A Capability Representation with Embedded Address and Nearly-Exact Bounds." Paper submitted to the ASPLOS 2000 committee for review.

[GRO01] Grossman, J.P. "Design and Evaluation of the Hamal Parallel Computer." Proposal for doctoral research submitted to MIT.

[IBMW1] "Blue Logic Cu-11 ASIC." IBM Publication SA14-2451-00, Copyright 2000. Publication may be obtained from http://www.chips.ibm.com.

[TSMW1] TSMC web page, 0.18 micron process summary. http://www.tsmc.com/technology/cl018.html

[MOSW1] Mosys web page. TSMC 0.13u process fast 1-T SRAM summary. http://www.mosys.com/1t_sram.html. Registration required to access design materials.

[PAP93] Papadopoulos, G. A. Boughton. Greiner, R. Beckerle, M. J. "*T: Integrated Building Blocks for Parallel Computing". Proceedings of the Conference on Supercomputing 1993. 1993. Pp. 623-635.

[NIK92] Nikhil, R.S. Papadopoulos, G.M. Arvind. "*T: A Multithreaded Massively Parallel Architecture." Proceedings of the 19th Annual International Symposium on Computer Architecture. 1992. Pp. 156-167.

[NUT91] Nuth, Peter R. Dally, William J. "A Mechanism for Efficient Context Switching." International Conference on Computer Design, 1991. 1991. Pp. 301-304.

[WEI98] Weissman, B. Gomes, B. Quittek, J.W. Holtkamp, M. "Efficient Fine-Grain Thead Migration with Active Threads." Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, 1998. (IPPS/SPDP 1998). IEEE 1998 Pp. 410 –414.

[HSI93] Hsieh, Wilson C. Wang, Paul. Weihl, William E. "Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems." Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, 1993. Pp. 239 – 248.

[CHA93] Chandra, Rohit. Gupta, Anoop. Hennessy, John L. "Data Locality and Load Balancing in COOL." Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, 1993. Pp. 249 – 259.

[CRO97] Cronk, D. Haines, M. Mehrotra, P. "Thread Migration in the Presence of Pointers." Proceedings of the Thirtieth Hawaii International Conference on System Sciences, 1997. IEEE, Vol. 1, 1997. Pp. 292 –298.

[NUT95] Nuth, Peter R. Dally, William J. "The Named-State Register File: Implementation and Performance." Proceeding of the First IEEE Symposium on High-Performance Computer Architecture. 1995. Pp. 4-13.

[TENW1] Xtensa Application Specific Microprocessor Solutions: Overview Handbook, a Summary of the Xtensa Data Sheet. Tensilica, Inc. Issue date: 2/2000.

[ARV90] Arvind. Nikhil, Rishiyur S. "Executing a Program on the MIT Tagged-Token Dataflow Architecture." IEEE Transactions on Computers. Vol 39, No 3. March 1990.

[CUL91] Culler, David E. Sah, Anurag. Schauser, Klaus Erik. von Eicken, Thorsten. Wawrzynek, John. "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine." Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1991. ACM, 1991. Pp. 164 – 175.

[NUT97] Nuttall, M. Sloman, M. "Workload Characteristics for Process Migration and Load Balancing." Proceedings of the 17th International Conference on Distributed Computing Systems, 1997. IEEE 1997. Pp. 133 –14.

[ROU96] Roush, Ellard T. Campbell, Roy H. "Fast Dynamic Process Migration." Proceedings of the 16th International Conference on Distributed Computing Systems, 1996. IEEE , 1996. Pp. 637 –645.

[NIK00] Nikolopoulos, Dimitrios S. Papatheodorou, Theodore S. Polychcronopoulus, Constantine, D. Labarta, Jesús. Ayguadé Eduard. "User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors." Proceedings of the International Conference on Parallel Processing, 2000. IEEE, 2000. Pp. 95-103.

[CHA94] Chandra, Rohit. Devine, Scott. Verghese, Ben. Gupta, Anoop. Rosenblum, Mendel. "Scheduling and Page Migration for Multiprocessor Compute Servers." Proceedings of ASPLOS VI, San Jose, CA. 1994.

[HSI95] Hsieh, W.C. Dynamic Computation Migration in Distributed Shared Memory Systems. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, September 1995. Available as MIT/LCS/TR-665.

[GAL] Galles, Mike. "Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER Chip." Whitepaper from the SGI website, since removed from the web. Contact bunnie@mit.edu for a copy of this paper.

[MCF97] McFarland, Grant W. CMOS Technology Scaling and Its Impact on Cache Delay. PhD Dissertation submitted to the Department of Electrical Engineering of Stanford University. June, 1997.

[DALJ98] Dally, William. Chien, Andrew. Fiske, Stuart. Horwat, Waldemar. Lethin, Richard. Noakes, Michael. Nuth, Peter. Spertus, Ellen. Wallach, Deborah. Wils, D. Scott. Chang, Andrew. Keen, John. "Retrospective: the J-Machine." 25 Years of the International Symposia on Computer Architecture (Selected Papers). 1998. Pp. 54-58.

[DAL98] Dally, William. Poulton, John W. Digital Systems Engineering. Cambridge University Press. 1998.

[NOA93] Noakes, M.D. Wallach, D.A. Dally, W.J. "The J-Machine Multicomputer: An Architectural Evaluation." Proceedings of the 20th Annual Symposium on Computer Architecture. 1993. Pp. 224-235.

[INTW1] Intel Developer Website. http://developer.intel.com.

[AMDW1] AMD Website. http://www.amd.com

[CPQW1] AlphaPoweredLinux Website. http://www.alphapoweredlinux.com/alpha21264.html.

[BAK90] Bakoglu, H.B. Circuits, Interconnections, and Packaging for VLSI. Addison-Wesley Publishing Company, Reading, MA. 1990.