Thesis Proposal:

Design and Evaluation of the Hamal Parallel Computer

J.P. Grossman
April 19, 2001

Supervisor: Tom Knight

Abstract

We wish to investigate design principles for general-purpose shared memory computers. Specific areas of interest are silicon efficiency, scalability, and RAM integration. Experiments will be performed using a cycle accurate simulator for the Hamal parallel computer. In this proposal we present an overview of the Hamal architecture and our plans for evaluating various mechanisms.

1 Introduction

Over the years there has been an enormous amount of hardware research in parallel computation. It is a testament to the difficulty of the problem that despite the large number of wildly varying architectures which have been designed and evaluated, there are few agreed-upon techniques for constructing a good machine. Even basic questions such as whether or not remote data should be cached are unanswered. This is in marked contrast to the situation in the scalar world, where many well-known hardware mechanisms are consistently used to improve performance (e.g. caches, branch prediction, speculative execution, out of order execution, superscalar issue, register renaming, etc.).

The primary reason that designing a parallel architecture is so difficult is that the parameters which define a "good" machine are extremely application-dependent. A simple physical simulation is ideal for a SIMD machine with a high processor to memory ratio and a fast 3D grid network, but will make poor utilization of silicon resources in a Beowulf cluster and will suffer due to increased communication latencies and reduced bandwidth. Conversely, a parallel database application will perform extremely well on the latter machine but will probably not even run on the former. Thus, it is important for the designer of a parallel machine to pick his fights early in the design process by identifying the target application space in advance

There is an obvious tradeoff involved in choosing an application space. The smaller the space, the easier it is to match the hardware resources to those required by user programs, resulting in faster and more efficient program execution. Hardware design can also be simplified by omitting features which are unnecessary for the target applications. For example, the Blue Gene architecture [IBM00], which is being designed specifically to fold proteins, does not support virtual memory [Denneau00]. On the other hand, machines with a restricted set of supported applications are less useful and not as interesting to end users. As a result, they are not cost effective because they are unlikely to be produced in volume. Since not everyone has $100 million to spend on an architecture, there is a need for commodity general-purpose parallel machines.

The term "general-purpose" is broad and can be further subdivided into three categories. A machine is general-purpose at the *application* level if it supports arbitrary applications via a restricted programming methodology; examples include Blue Gene [IBM00] and the J-Machine ([Dally92], [Dally98]). A machine is general-purpose at the *language* level if it supports arbitrary programming paradigms in a restricted run-time environment; examples include the RAW machine [Waingold97] and Smart Memories [Mai00]. Finally, a machine is general-purpose at the *environment* level if it supports arbitrary management of computation, including resource sharing between mutually non-trusting applications. This category represents the majority of parallel machines, such as Alewife [Agarwal95], Tera [Alverson90], The M-Machine ([Dally94], [Fillo95]), DASH [Lenoski92], FLASH [Kuskin94], and Active Pages [Oskin98]. Note that not all architectures fall cleanly into one of these categories. For example, Active Pages are general purpose at the environment level [Oskin99a], but not at the application level as only programs which exhibit regular, large-scale, fine-grained parallelism can benefit from the augmented memory.

The overall goal of the Hamal project is to investigate design principles for parallel architectures which are general-purpose at the environment level. Such architectures are inevitably less efficient than restricted-purpose hardware for any given application, but may still provide better performance at a fixed price due to the fact that they are more cost-effective. Focusing on general-purpose architectures, while difficult, is appealing from a research perspective as it forces one to understand mechanisms which support computation in a broad sense.

1.1 Specific Research Goals

In this thesis we will focus on three specific issues which represent fairly new areas of research in parallel computing: silicon efficiency, massive scalability, and RAM integration. The following sections will elaborate on each of these and explain why they are interesting and important subjects of investigation.

1.1.1 Silicon Efficiency

In current architectures there is an emphasis on executing a sequential stream of instructions as fast as possible. As a result, massive amounts of silicon are devoted to incremental optimizations such as those mentioned in the previous section. While these optimizations improve performance, they may reduce the architecture's **silicon efficiency**, when can be roughly defined as performance per unit area. However, this is immaterial since in a scalar machine the primary concern is overall performance.

Until recently the situation in parallel machines was similar. Machines were built with one processing node per die. Since, to first order, the overall cost of an N node system does not depend on the size of the processor die, there was no motivation to consider silicon efficiency. Now, however, designs are emerging which place several processing nodes on a single die ([Case99], [Diefen99], [IBM00]). As the number of transistors available to designers increases, this trend will continue with greater numbers of processors per die.

When a large number of processors are placed on each die, overall silicon efficiency becomes more important than the raw speed of any individual processor. With this observation we state our first research goal:

Evaluate novel and existing architectural mechanisms with respect to silicon efficiency.

1.1.2 Massive Scalability

Parallel shared memory machines with hundreds or thousands of processor/memory nodes have been built (e.g. [Dally98], [Laudon97], [SGI]); in the future we will see machines with millions [IBM00] and eventually billions of nodes. However, the memory systems of current parallel machines are not designed to be scaled to this extent. Two problems can be identified which are not addressed in a scalable manner by existing mechanisms: data location, and distributed object allocation.

Data location involves determining the physical location of a piece of data within the machine given a virtual address. Typically this information is folded into Translation Lookaside Buffers

(TLB's). This is effectively an attempt to store global information at each node. The TLB's become a cache for a data structure which must be kept globally consistent, and as such suffer from the same scaling difficulties as coherent data caches. Additionally, the size of this data structure grows linearly with the size of the machine for many applications while the TLB's do not, thus increasing the number of TLB misses which are already a serious performance problem in today's machines ([Chen92], [Huck93], [Jacob97], [Nagle93], [Saulsbury00], [Talluri92]). In order to support massive scaling, the memory system must be able to perform data location without storing global information at each node.

In large-scale shared memory systems, the layout of data in physical memory is crucial to achieving the maximum possible speedup. In particular, for many algorithms it is important to be able to allocate single objects in memory which are distributed across multiple nodes in the system. The challenge is to allow arbitrary single nodes to perform such allocations without any global communication or synchronization. A naïve approach is to give each node ownership of parts of the virtual address space that exist on all other nodes, but this makes poor use of the virtual address bits; an N node system would require 2logN bits of virtual address to specify location and ownership!

Our second research goal is to investigate memory system mechanisms which address these problems in a scalable manner. We state this as follows:

Develop a memory system which supports massive scaling.

1.1.3 RAM Integration

Over the past few years, several manufacturers have started offering processes that allow CMOS logic and DRAM to be placed on the same die. In its simplest form, this technique can be used to augment existing processor architectures with low-latency high-bandwidth memory [Patterson97]. A more exciting approach is to augment DRAM with small amounts of logic to extend its capabilities and/or perform simple computation directly at the memory. Several research projects have investigated various ways in which this can be done (e.g. [Oskin98], [Margolus00], [Mai00], [Gokhale95]). None of the proposed architectures are general-purpose at both the application and the environment level, due to restrictions placed on the application space and/or the need to associate a significant amount of application-specific state with large portions of physical memory. Our third and final research goal is therefore the following:

Investigate embedded DRAM augmentations which support environment-level general-purpose computing.

1.2 Approach

The vehicle for our research will be a flexible cycle-accurate simulator for the Hamal architecture. Hamal is a parallel shared-memory machine which integrates a number of new and existing architectural ideas; it will be described in more detail in section 2. In section 4 we will propose a set of micro- and macro-benchmarks which exercise the features under study. Determining the performance of these benchmarks across various hardware configurations and machine loads will allow us to evaluate specific aspects of the Hamal design. When combined

with area estimates for hardware implementations, these performance numbers will also show how selected mechanisms impact the silicon efficiency of the architecture. Finally, the benchmarks will provide us with an evaluation of the system as a whole; of particular interest will be the degree to which we were successful in designing a machine which is general-purpose at the environment level.

1.3 Main Contributions

The first major contribution of this thesis will be the presentation of novel memory system features to support a scalable, efficient parallel system. We will show how to address the problems of data location and distributed object allocation discussed in section 1.1.2. We will present a capability format which supports pointer arithmetic and nearly-tight object bounds without the use of capability or segment tables. We will describe a flexible scheme for synchronization within the memory system. We will show how forwarding pointers can be supported with minimal overhead. Our focus throughout will be on enhancing the performance, silicon efficiency and scalability of an architecture.

The second contribution will be to evaluate the silicon efficiency of certain existing architectural mechanisms. Mechanisms of interest include Very Long Instruction Words (VLIW), hardware multithreading and hardware page tables. In some cases there is extensive literature on the absolute performance advantages of these mechanisms under various conditions, but little or no quantitative evaluations of the performance-area tradeoff. We will be specifically interested in determining the parameters of a given mechanism which maximize silicon efficiency (e.g. the number of hardware contexts).

The third and final major contribution will be the complete description and evaluation of a general-purpose embedded-memory parallel computer. This will provide a design point against which other general-purpose architectures can be compared. Additionally, the advantages and shortcomings of the Hamal architecture will further our understanding of how to build a good parallel machine.

A fourth and minor contribution will be the presentation of a C++ framework for cycle-based hardware simulation. This contribution is more of a side effect of our research methodology than a direct result of our research goals. Nonetheless, we feel that it is significant as software simulation plays an important role in hardware design, and simulators are often difficult to write and debug. The framework is designed to enhance modularity and ease debugging; its primary feature is support for timestamped values.

1.4 Putting on the Blinders

As stated previously, the focus of this work will be on scalability, silicon efficiency and overall performance. It is worth mentioning three areas of research that we are specifically *not* interested in. The first of these is power. Simply put, we don't care how much power our machine uses. While this would affect the cost-effectiveness of an actual implementation due to the need for liquid cooling, it does not present any fundamental problems. One can always deal with hotter chips using faster water [Knight00]. The second area of research that we ignore is fault tolerance. Built-in fault tolerance is essential for any massively parallel machine which is

to be of practical use (a million node computer is a great cosmic ray detector). However, the design issues involved in building a fault tolerant system are for the most part orthogonal to the issues which are under study. Our simulations will therefore make the simplifying assumption of perfect hardware. The third area of research that we explicitly disregard is network design. Certainly a good network is of fundamental importance, and the choice of a particular network will have a first order effect on the performance of any parallel machine. However, contributing to the already large body of network research is beyond the scope of this thesis. Of course, it is still important to understand the impact of the network on the machine; in particular results obtained concerning specific architectural mechanisms should as much as possible be independent of the actual network used. Accordingly, all simulations will be carried out using three different network models: a simple 3D mesh, a magic zero-latency full crossbar network, and a more realistic fat tree network with nearest-neighbour connections at the leaves.

2 The Hamal Parallel Computer

The Hamal architecture will be used as a test bed to evaluate a number of different mechanisms. Our primary research tool will be a cycle-accurate simulator of a complete machine. In this section we will describe the architecture and discuss some of the key features.

2.1 Overview

The Hamal Architecture consists of a large number of identical processor/memory nodes connected by a fat tree network. The design intends for multiple nodes to be placed on a single die, which provides a natural path for scaling with future process generations (by placing more nodes on each die), and also makes silicon efficiency a primary concern. Each node contains a 129 bit multithreaded VLIW processor, 512KB of data memory, 128KB of code memory, and a network interface (Figure 1). Each 128 bit memory word and register in the system is tagged with a 129th bit to distinguish pointers from raw data. Shared memory is implemented transparently by the hardware.

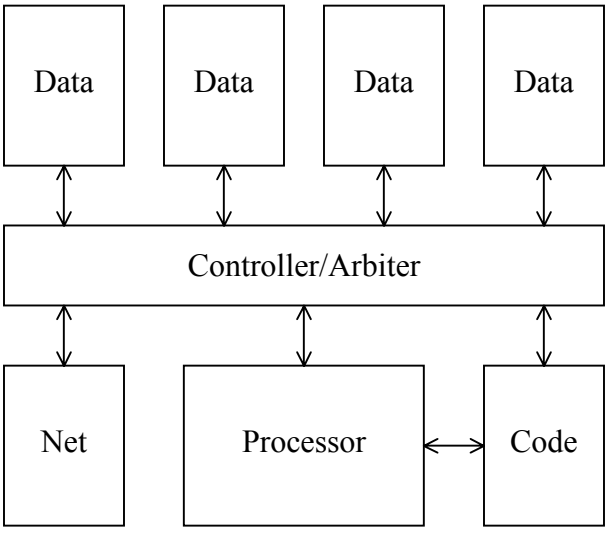

Figure 1: The Hamal processing node

The memory sizes given above are preliminary estimates; in a hardware implementation they would be chosen so that processor and memory consume roughly equal amounts of silicon. The rationale for this is that while any given processor to memory ratio will necessarily be efficient for some applications and inefficient for others, the worst case inefficiency for an even division is roughly a factor of two. An important remark is that the ratio is *fixed*; processor and memory are inseparable, and adding memory to the system necessary adds processors. In addition to reducing the number of distinct components in the system, this design improves the asymptotic behavior of many problems [Oskin98].

There are no data caches in the system for a number of reasons. First, with on-die DRAM it is already possible to access local memory in only a few cycles. A small number of hardware contexts can therefore tolerate memory latency and keep the hardware busy at all times. Second, caches consume large amounts of silicon area, which may negatively impact the silicon efficiency of the architecture. Third, designing a coherent cache for a massively parallel system is an extremely difficult and error-prone task.

2.2 Memory System

One cannot over-emphasize the importance of a properly designed memory system. In a shared memory parallel computer, the memory model and its implementation have a direct impact on system performance, programmability and scalability. The Hamal memory system has been designed to address our specific goals of silicon efficiency, massive scalability and RAM integration.

2.3 Memory Semantics

Sequential consistency presents a natural and intuitive shared memory model to the programmer. Unfortunately, it also severely restricts the performance of many parallel applications ([Gharach91], [Zucker92], [Chong95]). This is due to the fact that no memory operation from a given thread may proceed until the effect of every previous memory operation from that thread is globally visible in the machine. This problem becomes worse as machine size scales up and the average round trip time for a remote memory reference increases.

In order to maximize program efficiency, Hamal makes no guarantees concerning the order in which references to different memory locations complete. Memory operations are explicitly split-phase; a thread continues to execute after a request is injected into the system, and at some unknown time in the future a reply will be received. The hardware will only force a thread to stall in three circumstances:

1. The result of a read is needed before a reply containing the value is received
2. There is a RAW, WAR or WAW hazard with a previous memory operation that has not completed
3. The hardware table used to keep track of incomplete memory operations is full

A *wait* instruction is provided to force a thread to stall until all outstanding memory operations have completed. This allows release consistency [Gharach90] to be efficiently implemented in software.

2.3.1 Capabilities

If a machine is to support environment-level general purpose computing, one of the first requirements of the memory system is that it provide a protection mechanism to prevent applications from reading/writing each other's data. In a conventional system, this is accomplished by providing each process with a separate virtual address space. While such an approach is functional, it has three significant drawbacks. First, a process-dependent address translation mechanism dramatically increases the amount of machine state associated with a given process (page tables, TLB entries, etc), which increases system overhead and is an impediment to fine-grained multitasking. Second, data can only be shared between processes at the page granularity, and doing so requires some trickery on the part of the operating system to ensure that the page tables of the various processes sharing the data are kept consistent. Finally, this mechanism does not provide security within a single context; a program is free to create and use invalid pointers.

These problems all stem from the fact that in most architectures there is no distinction at the hardware level between pointers and integers; in particular a user program can create a pointer to an arbitrary location in the virtual address space. An alternate approach which addresses these problems is the use of unforgeable capabilities ([Dennis65], [Fabry74]). Capabilities allow the hardware to guarantee that user programs will make no illegal memory references. It is therefore safe to use a single shared virtual address space with greatly simplifies the memory model.

In the past capability machines have been implemented using some form of capability table ([Houdek81], [Tyner81]) and/or special capability registers ([Abramson86], [Herbert79]), or even in software ([Anderson86], [Chase94]). Such implementations have high overhead and are an obstacle to efficient computing with capabilities. However, in [Carter94] a capability format is proposed in which all relevant address, permission and segment size information is contained in a 64 bit data word. This approach obviates the need to perform expensive table lookup operations for every memory reference and every pointer arithmetic operation. Additionally, the elimination of capability tables allows the use of an essentially unbounded number of segments (blocks of allocated memory); in particular object-based protection schemes become practical. Their proposed format requires all segment sizes to be powers of two and uses six bits to store the base 2 logarithm of the segment size, allowing for segments as small as one byte or as large as the entire address space.

Restricting segment sizes to powers of two causes three problems. First, since the size of most objects is not a power of two, there will be a large amount of internal fragmentation within the segments. This reduces the likelihood of detecting pointer errors in programs as pointers can be incremented past the end of objects while remaining within the allocated segment. Second, this fragmentation causes the apparent amount of allocated memory to exceed the amount of in-use memory by as much as a factor of two. This can seriously impact the performance of system memory management strategies such as garbage collection. Finally, the alignment restriction may cause a large amount of external fragmentation when objects of different size are allocated. As a result, a larger number of physical pages may be required to store a given set of objects.

Hamal uses 129 bit capabilities which are broken down into 64 bits of address, 64 bits of capability information (segment size, permissions, etc), and a single tag bit to distinguish pointers from raw data. As in [Carter94], capabilities may be mixed freely with data. To allow for more flexible segment sizes, 11 bits are used to specify segment size. 6 bits specify the block size (B), and 5 bits specify the length (L) of the segment (in blocks); the segment size is $L \cdot 2^B$. This representation reduces internal fragmentation to less than 6% in the worst case. An additional 5 bits are used to specify the current block index (K), that is, which block within the segment a given capability points to. This allows the segment's base address to be recovered from a pointer to any location within the segment. Note that the format in [Carter94] can be viewed as a special case of this format in which L = 1 and K = 0 for all capabilities.

Our capability format also includes an increment-only bit (I) and a decrement-only (D) bit. It is an error to add a negative offset to a capability with I set, or a positive offset to a capability with D set. These bits provide a simple mechanism for implementing exact object bounds and sub-object security.

2.3.2 Virtual memory

The memory model of early computers was simple: memory was external storage for data; data could be modified or retrieved by supplying the memory with an appropriate physical address. The model was directly implemented in hardware by discrete memory components. This simplified view of memory has long-since been replaced by the abstraction of virtual memory, yet the underlying memory components have not changed. Instead, complexity has been added to processors in the form of logic which performs translations from sophisticated memory models to simple physical addresses.

There are a number of drawbacks to this approach. The overhead associated with each memory reference is large due to the need to look up page table entries. All modern processors make use of translation lookaside buffers (TLB's) to try to avoid the performance penalties associated with these lookups. A TLB is essentially a cache, and as such provides excellent performance for programs that use sufficiently few pages, but is of little use to programs whose working set of pages is large. Another problem common to any form of caching is the "pollution" that occurs in a multi-threaded environment; a single TLB must be shared by all threads which reduces its effectiveness and introduces a cold-start effect at every context switch. Finally, as mentioned in section 1.1.2, in a multiprocessor environment the TLB's must be kept globally consistent which places constraints on the scalability of the system.

The Hamal architecture addresses these problems by implementing virtual memory at the memory rather than at the processor. Associated with each bank of DRAM is a hardware page table with one entry per physical page. These hardware page tables are similar in structure and function to the TLB's of conventional processors. They differ in that they are persistent (since there is a single shared virtual address space) and complete; they do not suffer from pollution or cold-starts. They are also slightly simpler from a hardware perspective due to the fact that a given entry will always translate to the same physical page.

2.3.3 Hardware LRU

Most operating systems employ a Least Recently Used (LRU) page replacement policy. Typically the implementation is approximate LRU rather than exact LRU, and some amount of work is required by the operating system to keep track of LRU information and determine the LRU page. In the Hamal Architecture, a systolic array is added to each DRAM bank to maintain exact LRU information. A single instruction provides this information to the operating system. This allows page faults to be serviced with minimal overhead.

2.3.4 Sparse Objects

The problems of data location and distributed object allocation discussed in section 1.1.2 are both dealt with using *sparse objects*. In this scheme, the upper virtual address bits are used to identify a physical node. This defines a fixed mapping from virtual addresses to physical nodes, which allows data to be located without storing any global information at each node.

Distributed objects are allocated in the same manner as node-local objects, but a single bit in the capability is set to indicate that the object is *sparse*. Initially, a sparse object is explicitly allocated only on its home node, but it is implicitly allocated on *all* nodes. From the allocating node's point of view, the same segment (i.e. the same size and base address) appears to be allocated on each node; we refer to each of these segments as a *facet* of the object. However, the same segment cannot in reality be allocated on each node without global communication to avoid address conflicts.

The key mechanism which enables sparse objects is a translation table which exists at the boundary of each processing node to translate local sparse object addresses to/from global segment unique identifiers (GSUID) with offsets. When a sparse pointer moves from a node to the network, it is first decomposed into a segment base address and an offset. The base address is used to look up the segment's GSUID in the translation table. If no entry exists in the table, which can only occur the first time a sparse pointer leaves its home node, a new GSUID is created which consists of the node identifier and a node-local unique identifier. When a GSUID arrives at a node, it is used to look up the segment's local base address in the translation table. If no entry exists in the table, which occurs the first time a node sees a pointer to a given sparse object, then a local facet is allocated and the base address is entered into the table.

The term "sparse object" reflects the fact that object facets are lazily allocated and may exist on only a small portion of the system's nodes. Note that the implementation requires global information to be stored at individual nodes which can potentially affect the scalability of the system. Another issue is the performance degradation which occurs if the working set of GSUIDs on some node exceeds the size of the hardware translation table.

In order for distributed objects to be useful, it must be possible to lay out an object across physical nodes in a flexible manner without performing excessive computation on indices. To compensate for a somewhat rigid mapping from virtual addresses to physical nodes, a hardware *swizzle* instruction is provided which combines a 64 bit operand with a 64 bit mask to produce a 64 bit result by right-compacting the operand bits corresponding to 1's in the mask, and left-compacting the operand bits corresponding to 0's in the mask. The swizzle instruction has a

number of potential uses; the specific use which concerns us here is the ability to implement an address centrifuge [Scott96] in software using a single hardware instruction.

2.3.5 Atomic Memory Operations

With the ability to place logic and memory on the same die, there is a strong temptation to engineer "intelligent" memory by adding some amount of processing power. However, in systems with tight processor/memory integration there is already a reasonably powerful processor next to the memory; adding an additional processor would do little more than waste silicon and confuse the compiler. The processing performed by the memory in the Hamal architecture is therefore limited to simple single-cycle atomic memory operations such as addition, maximum and boolean logic. These operations are useful for efficient synchronization and are similar to those of the Tera [Alverson90] and CrayT3E [Scott96] memory systems.

2.3.6 Trap Bits

Three trap bits (T, U, V) are associated with every 128 bit memory word. The T bit specifies that any reference to the memory location will cause a trap. It is available to the operating system and can be used to implement mechanisms such as data breakpoints and forwarding pointers. The U and V bits are available to user programs to enrich the semantics of memory accesses via customized trapping behaviour. Each instruction that accesses memory is prefixed by eight bits which specify how U and V are interpreted and/or modified. For each of U and V, two bits specify behaviour; the possibilities are ignore the trap bit, trap on set, and trap on clear. For each of U and V, two bits specify how the trap bit is modified; the possibilities are set, clear, toggle, and no change.

2.3.7 SQUIDs

Forwarding pointers are a conceptually simple architectural mechanism that allow references to a memory location to be transparently forwarded to another location. Known variously as "invisible pointers" [Greenblatt74], "forwarding pointers" [Moon85] and "memory forwarding" [Luk99], they are relatively easy to implement in hardware, and are an invaluable tool for safe data compaction ([Luk99], [Brown99]). However, they have the significant drawback that they introduce aliasing; it is possible for two different pointers to resolve the same word in memory. This aliasing has an associated run-time cost as direct pointer comparisons are no longer a safe operation; some mechanism must be provided for determining the final addresses of the pointers being compared.

In the Hamal architecture, this problem is addressed using **SQUIDs** (**S**hort **Q**uasi-**U**nique **ID**'s). Each object is assigned a short random tag, stored in pointers to the object, which is similar in role to a UID, but is not necessarily unique. Two pointers can be efficiently compared by examining their base addresses, offsets and squids. If the base addresses are the same then the pointers point to the same object, and the pointers are the same if and only if they have the same offset into the object. If the squids are different then they point to different objects. It is only in the case that the base addresses are different but the squids and offsets are the same that it is necessary to perform expensive dereferencing operations to determine whether or not the final addresses are equal.

It can be argued that this latter case will be rare. It occurs in two circumstances: either the pointers reference different objects which have the same squid, or the pointers reference the same object through different levels of indirection. The former of these occurs with probability 2^{-n} (using n-bit squids). The latter is application dependent, but we note that (1) applications tend to compare pointers to different objects more frequently than they compare pointers to the same object, and (2) the results of the simulations in [Luk99] indicate that it is reasonable to expect the majority of pointers to migrated data to be updated, so that two pointers to the same object will usually have the same level of indirection.

2.4 Processor Design

Each processor-memory node contains a 129-bit multithreaded VLIW processor. There are four hardware contexts; of these, context 0 is reserved for event handling (a special *poll* instruction is provided to obtain events from a hardware event queue), and contexts 1-3 are available for running user programs. Instructions may be issued from a different context on each cycle, and instructions from multiple contexts may complete in a given cycle.

There are 32 128-bit general purpose registers, r0-r31. Each register is tagged with a single additional pointer bit indicating whether or not it contains a pointer. Each register may be addressed as a single 128 bit register (e.g. r7), two 64 bit registers using the suffixes ‘x’, ‘y’ for the low, high bits respectively (e.g. r7x, r7y), or four 32 bit registers using the suffixes ‘a’, ‘b’, ‘c’, ‘d’ for the low, low-middle, high-middle, high bits respectively (e.g. r7a, r7b, r7c, r7d). See Figure 2.

r3		r2y	r2x		r1y	r1b	r1a		r0d	r0c	r0b	r0a
r7		r6			r5				r4			
r11		r10			r9				r8			
r15		r14			r13				r12			

⋮

Figure 2: General Purpose Registers

Allowing registers to be accessed in this multi-granular fashion makes it possible to use a single register set for both 128 bit pointers and data without suffering from internal fragmentation. Additionally, many arithmetic operations operate in parallel on four sets of 32 bit operands, providing compilers with the opportunity to exploit fine grained SIMD-style parallelism within a program.

Each VLIW instruction group consists of three instructions and an immediate. One instruction is an arithmetic instruction which specifies up to two source operands and a destination register. One instruction is a memory instruction which specifies up to three source operands (address, index, data) and a destination register. The last instruction is a control flow instruction which specifies zero or one operands. Predicated execution is supported; each instruction within an instruction group can be independently predicated on the value (true or false) of 31 hardware-defined predicates.

3 Simulation

Software simulation is an essential and also challenging part of hardware design. Ideally, a software simulator should be modular, fast, and, of course, correct. The choice of language is important; writing the simulator in an efficient object-oriented language such as C++ can help meet the goals of speed and modularity. However, some additional strategies are required to facilitate the elimination of software bugs.

3.1 Sources of Error

Typically, a modular simulator is written by using classes to model hardware components. Each class is given member variables corresponding to the component's inputs and outputs. Simulation proceeds by alternately updating components (using the inputs and internal state to generate outputs and a new internal state) and propagating data between components by copying outputs to inputs according to the connections in the hardware being modeled. In such a simulation there are three common sources of error:

Invalid Outputs. The update routine(s) for a component may neglect to set one or more outputs in some cases, resulting in garbage or stale values being propagated to other components.

Missing Connections. One or more of a component's inputs may never get set.

Bad Update Order. When the simulation involves components with combinational paths from inputs to one or more outputs, the order in which components are updated and outputs are copied to inputs becomes important. An incorrect ordering can have the effect of adding or deleting registers at various locations.

While careful coding can avoid these errors in many cases, it is simply impossible to write a large piece of software without introducing bugs. In addition, these errors are particularly difficult to track down as in most cases they produce silent failures which go unnoticed until some unrelated output value is observed to be incorrect. The programmer is often required to spend enormous amounts of time finding the exact location and nature of the problem.

3.2 A C++ Framework for Efficient Hardware Simulation

We have developed *Sim*, a C++ framework for constructing cycle-based hardware simulators. Sim provides the programmer with three abstractions: components, nodes and registers. A component is a C++ class which is used to model a hardware component. In debug builds, Sim automatically constructs a named hierarchy of components so that error messages can give the precise location of the fault in the simulated hardware. A node is a container for a value which supports connections and, in debug builds, timestamping. Nodes are used for all component inputs and outputs. Registers are essentially D flip-flops. They contain two nodes, D and Q; on the rising clock edge D is copied to Q.

Simulation proceeds in three phases. In the construction phase, all components are constructed and all connections between inputs and outputs are established. When an input/output node in one component is connected to an input/output node in another component, the two nodes

become synonyms, and writes to one are immediately visible to reads from the other. In particular, it is no longer necessary to explicitly copy values from outputs to inputs at each timestep in the simulation. In the initialization phase, Sim prepares internal state for the simulation and initial values may be assigned to component outputs. Finally, the simulation phase consists of alternating calls to the top-level component's Update function (to simulate combinatorial evaluation) and a global Tick function (which simulates a rising clock edge).

Sim's most useful feature is timestamped nodes. In debug builds, each time a node is written to it is timestamped, and each time a node is read the timestamp is checked to ensure that the node contains valid data. This mechanism detects all three of the errors described in the previous section. When an invalid timestamp is encountered, a warning message pinpoints the component and input/output which generated the error. This can speed up the debugging process by an order of magnitude, allowing the programmer to detect and correct errors in minutes that would otherwise require hours of tedious work.

3.3 Evaluation

There are three important points of comparison between simulations written with Sim and simulations written in straight C++: the ease of programming/debugging, the performance of the simulation, and the amount of memory required. The first of these is difficult to quantify, but it is clear that Sim provides important debugging support. The latter comparisons can be made more precisely by implementing a set of hardware benchmarks in both Sim and plain C++, then collecting performance and memory usage statistics. The following benchmarks will be implemented in this manner:

- A 3-tap linear feedback shift register
- A systolic array for performing least recently used computation
- A 2D torus network with wormhole routing
- An FPGA
- A simple pipelined microprocessor

4 Benchmark Suite

An architecture cannot be evaluated without a good set of benchmarks. The following benchmarks have been chosen to stress all aspects of the Hamal architecture which are under study. In addition, they represent a variety of applications which can require enormous computing resources and will be scalable to massively parallel machines.

4.1 FFT

No benchmark set is complete without a fast fourier transformation. FFT tests both the computational power of the processors and the effectiveness of the network. Of interest in the Hamal architecture will be the efficiency with which the FFT algorithm can be parallelized.

4.2 Quicksort

Quicksort is an ideal benchmark for parallel computers. It is easy to program and it is one of the most fundamental algorithms in computer science. It challenges the hardware's ability to deal with large distributed vectors, data redistribution, and synchronization. Specific to the Hamal architecture, it will be a good benchmark for dynamic sparse objects.

4.3 Matrix Manipulation

Matrices are at the heart of nearly all linear algebra computations. We will focus on multiplication, LU decomposition, determinants and inverses. These operations require recursion and dynamic objects; they exhibit regular computation within small inner loops.

4.4 Sparse Matrix Manipulation

Large sparse matrices play an important role in many different applications. Creating a sparse matrix will test Hamal's sparse object, synchronization and allocation mechanisms. Multiplying two sparse matrices will stress the communications network. Sparse matrices provide a good combination of numerical computation and dynamic data structures.

4.5 Database

Managing large data sets is an important capability for a general purpose parallel computer. Our database benchmark will implement the *query* and *join* operations. For medium sized databases, this will primarily test communication and synchronization. For larger databases, it will also test Hamal's page management mechanisms.

4.6 Memory Streaming

Memory streaming is a technique to tolerate long memory latencies by actively prefetching data before it is needed. In the past, special purpose hardware mechanisms have been proposed to support memory streaming (e.g. [Smith82], [Hwang01]). Hamal, by contrast, provides a general purpose synchronization mechanism (memory trap bits) which can be used to implement memory streaming in software. A memory streaming benchmark will test the effectiveness of these mechanisms for one of their potential uses.

4.7 Hardware Simulation

Netlist simulations typically contain large amounts of fine-grained parallelism, but are difficult to parallelize efficiently due to the overhead of synchronization and thread management. The Hamal architecture contains a number of mechanisms designed to reduce these overheads; a hardware simulation benchmark will put these mechanisms to the test.

4.8 Ray Tracing

Ray tracing is easy to parallelize as each ray in the scene is independent of all others. Load balancing is a challenge, however, since in complicated scenes the path that a given ray takes can

be fairly random. Ray tracing will stress communication and mechanisms for handling memory latency. Additionally (not directly related to this thesis), it is a good benchmark for testing language-level parallelism primitives.

4.9 Cellular Automata

In contrast to the irregular parallelism and communication of ray-tracing, cellular automata is extremely regular in nature and requires only local communication. Since the computation requirements of cellular automata are typically small, the main factor determining the speed of the simulation will be the effective bandwidth to memory and/or the bandwidth of local communication. More precisely, for large simulations most cells have no neighbours on remote nodes, and memory bandwidth dominates. For small simulations, a larger fraction of cells have remote neighbours and the bandwidth of local communication dominates.

4.10 N body simulations

N body problems with pairwise interactions are of fundamental importance in a wide range of physical simulations, ranging from galaxy collisions to protein folding. For large simulations with many bodies per processing node, communication dominates. For smaller simulations with fewer bodies than processing nodes, parallelism is implemented on the level of pairs rather than bodies, and both computational power and synchronization become important.

4.11 Point Sample Rendering

In the past few years there has been an interest in using point samples to model and render complex objects ([Grossman98], [Pfister00], [Rusink00]). Point sample rendering is particularly well suited to implementation on a general purpose parallel machine as each point sample can be rendered independently. Rendering an object involves retrieving a large database of point samples, performing hierarchical culling, transforming points to screen space and storing them in a Z-buffer, and performing lighting in screen space. It is a good test of memory bandwidth, communication and dynamic threads.

4.12 Lisp Interpreter

In a sense, a lisp interpreter is many benchmarks rolled into one. Resolving names quickly is a parallel hashing benchmark. Executing lisp programs efficiently requires fast type checking and object allocation (fine-grained ILP and dynamic object benchmarks). Finally, free memory is reclaimed via garbage collection, an important benchmark in itself. A lisp interpreter will stress the various memory mechanisms of the Hamal architecture.

4.13 Multiprogramming

Since the Hamal architecture is intended to provide a platform for general-purpose computing at the environment level, it is important to evaluate its performance under a multiprogrammed workload. This can be accomplished fairly easily by running several benchmark programs (not necessarily distinct programs) at the same time.

5 Evaluation

The Hamal architecture integrates a number of mechanisms and design features which we wish to evaluate. Our general approach will be to run benchmark programs on a cycle accurate simulator with varying design parameters. The following sections provide further details on how we intend to evaluate the architecture.

5.1 Hardware page tables

Conventional machines use TLB's to perform virtual to physical address translation at the processor; in the Hamal architecture this translation is performed at the memory by hardware page tables. The benefits of hardware page tables are clear: they simplify processor design, they eliminate the need for a hardware or software TLB miss handler, they avoid TLB contention issues when multiple memory banks on a node are accessed simultaneously, and they support arbitrary system scaling. We will therefore be primarily interested in evaluating their cost.

The main cost of hardware page tables is the silicon required to implement them. This cost is clearly application-independent. It can be estimated fairly easily by laying out a hardware page table in a 0.18μ CMOS process. We can express this cost as an area-equivalent amount of memory.

A secondary cost of any virtual memory system is the software page tables which must be maintained. This cost is application-dependent. In a conventional system, complete page tables must be managed by the operating system to enable the location of any virtual page in memory or on disk. With hardware page tables it is no longer necessary to store location information for in-core pages, but locating pages on disk is still an issue. However, we will assume that an intelligent secondary-storage system manages the storage and retrieval of virtual pages. Thus, the only information that needs to be stored in the software page tables is whether or not a given virtual page exists. This information is used when a page fault occurs to decide whether the page needs to be created or retrieved from secondary storage. This significant simplification to the software page tables represents a potentially considerable savings compared to the full page tables required in conventional systems. We will quantify this savings, also measured as an amount of memory, across the applications in the benchmark suite. It is our hypothesis that, on average, this savings will outweigh the area cost of hardware page tables.

5.2 Hardware LRU

As with hardware page tables, there is both a cost and a savings associated with hardware LRU. The cost is again silicon which we will measure by laying out a systolic LRU cell. The primary savings is the operating system code which is *not* required to implement LRU page replacement. Note that neither of these are application-dependent. There is also a small performance advantage associated with hardware LRU due to the fact that the operating system code which does not exist is (obviously) never executed. This frees up processor cycles which may be used by other threads.

5.3 Capability format

The segment size representation in our capability format can be parameterized by the size in bits of the L field. With zero bits, the representation is identical to that of guarded pointers [Carter94]. The main point of evaluation will be the amount of internal and external fragmentation as a function of the size of the L field. The relevant benchmark applications are those that make significant use of dynamically allocated objects, namely quicksort, matrices, database, and lisp interpreter.

The benefits of the increment only and decrement only bits in our capability format are difficult to quantify. Our evaluation will be purely qualitative; we will provide examples of their applicability to exact object bounds and sub-object security.

5.4 Squids

Squids are a simple hardware mechanism which are intended to reduce the run time costs associated with forwarding pointer aliasing. Specifically, in the common case they allow pointers to be compared without being dereferenced in an environment that supports transparent forwarding pointers. Squids can be evaluated in a straightforward manner by running benchmark programs with various numbers of bits allocated for squids (where zero bits represents a machine configuration that does not support squids). This will allow us to measure the precise performance gains provided by squids on the Hamal architecture. It is also possible to approximately quantify their advantages in a more general setting by instrumenting programs run in a standard environment to collect statistics on how frequently pointers are compared and what fraction of those comparisons would not be resolved by squids. Together with a model for the cost of dereferencing pointers, this would provide us with an estimate of the the run-time savings on an arbitrary architecture (it would be an estimate only due to the unpredictability of cache effects).

5.5 Sparse Objects

Sparse objects allow distributed segments to be allocated efficiently with no global communication. The primary cost, and the main point of evaluation, is the translation table required at node boundaries to translate between GSUIDs and local virtual addresses. Of interest is the tradeoff between table size and performance. Additionally, we will assess the impact of sparse objects on system scalability by measuring the required table size as the number of processor nodes is increased exponentially.

5.6 VLIW

The decision to use VLIW processors at each node was based on considerations of silicon efficiency. VLIW offers many of the advantages of dynamic superscalar execution without the associated complexity and area overhead. However, it comes at the cost of increased code size. Thus, while area is saved in the processor, more area is required for instruction memory, and the overall area of a processor-memory node may increase. Because of this, it is not clear a priori whether or not a VLIW processor will result in better silicon efficiency than a vanilla single-instruction issue processor. To answer this question, we will use the benchmark programs to

quantify both the necessary increase in instruction memory size and the average performance gain afforded by VLIW over a single-instruction issue processor.

5.7 Memory trap bits

Placing T, U and V trap bits in the memory system is one of the ways in which Hamal augments embedded DRAM with small amounts of logic. Because only a small subset of the application space will make use of these trap bits, it is difficult to evaluate them in a precise quantitative manner. Our approach will be to present potential uses for the trap bits and then evaluate them based on these specific applications. For example, the 'T' bits can be used to implement forwarding pointers (lisp benchmark), and the UV bits can be used to implement synchronization in the memory streaming benchmark.

5.8 Hardware Multithreading

Hardware multithreading is a technique for improving processor utilization by allowing pipeline bubbles in one thread of execution to be filled by instructions from another thread. We wish to evaluate hardware multithreading from the point of view of silicon efficiency; in particular we want to find the number of hardware threads for which silicon efficiency is maximized. To accomplish this, we will estimate the area overhead of additional contexts, and we will measure benchmark performance for varying numbers of contexts.

6 Previous Work

There is an enormous body of literature related to the Hamal architecture and the specific research questions that we wish to address. As is so often the case in computer architecture, a large part of the Hamal design is simply a new combination of old ideas. In this section we will attempt to summarize some of the most relevant previous work.

6.1 RAM Integration

The easiest way to integrate logic and memory is to add some basic data processing capabilities to memory and expose these capabilities to a host processor in a SIMD manner so that a restricted set of applications may be accelerated. One of the earliest proposals of this sort appears in [Kowarik90], where it is suggested that DRAM be augmented with simple boolean and associative matching operations. In [Gokhale95] the Terasys prototype is described which adds a 1-bit ALU to each column of memory. A fully populated system with 32K single bit processors running at 10MHz is controlled by a single Sparc-2 processor and is capable of performing 3.2×10^{11} bit operations per second. Active Pages support a broader spectrum of computation by associating a 256 logic element reconfigurable array [Oskin98] or a simple VLIW processor [Oskin99] with each 512KB page of data memory. The difficulty with these approaches to RAM integration lies in determining how a given application should be partitioned between the host processor and the augmented memory. Typically this partition has been performed by hand, but recently there has been some progress towards automating this process within a compiler [Lee01].

Another approach to RAM integration is to focus on bandwidth rather than computation; on-chip DRAM increases memory bandwidth by an order of magnitude. The Berkeley IRAM project makes use of this bandwidth by combining DRAM with a vector processor on a single die [Kozyrakis97]. In [Margolus00], 64 simple processing elements are associated with each 4Mb block of DRAM in an architecture tailored for spatial-lattice computation. By making full use of the available memory bandwidth, it is estimated that this architecture will provide a 10,000-fold speedup over the CAM-8 lattice gas computer [Margolus96].

Some strides towards RAM integration for general-purpose computing are made in [Mai00], in which a highly reconfigurable processor-memory architecture is described. The proposed architecture consists of a 2D array of tiles, where each tile contains a 64 bit processor, a network interface, and 128KB of SRAM. It is shown that this architecture can be used to emulate two other hardware designs with reasonable efficiency. However, it is not clear how the architecture can be programmed directly.

A slightly different idea is to augment DRAM with SRAM rather than logic in order to speed up memory accesses. In [Ward88] it is observed that DRAM parts already contain a static row buffer which acts as a small cache; adding a few extra row buffers raises the hit ratio and improves the performance of the DRAM. In [Glaskow99] the MoSys 1T-SRAM memory is described which takes this idea to an extreme. An SRAM cache is associated with multiple banks of DRAM. When an access hits the cache, a row from each DRAM bank can be refreshed in parallel. The size of the cache is chosen to guarantee that all DRAM rows will eventually be refreshed, which eliminates wait states and allows the DRAM macro to have an SRAM interface.

6.2 Silicon Efficiency

Most of the previous work related to silicon efficiency has focused on reducing complexity without sacrificing performance rather than explicitly maximizing efficiency. For example, in [Ahuja95] it is shown that many bypasses can be omitted without seriously impacting performance, which reduces wiring requirements and saves area. In some cases researchers have investigated the area-performance tradeoff more directly; in [Conte93] various functional unit mixes are analyzed in order to maximize performance within a given area, and in [Oskin99b] it is found that a VLIW active page implementation achieves the same performance as a reconfigurable logic implementation with a substantial reduction in area.

One mechanism which has received a great deal of attention is dynamic out-of-order execution. This is a well-known technique for accelerating scalar programs which has been incorporated into most commercial processors. However, it is also extremely costly; one implementation of two 28 entry reorder buffers required 850,000 transistors [Gaddis96]. In [Palacharla97] it is shown that the expensive issue window can be replaced with a small number of simpler FIFO buffers without seriously impacting performance. In [Hily99] it is found that dynamic out-of-order execution has little impact on overall performance in a processor that supports simultaneous multithreading. A mechanism is proposed in [Grossman00] for achieving out-of-order execution with in-order issue logic by allowing the compiler to specify explicit delays for instructions.

6.3 Hardware Page Tables

Hardware Page Tables were first proposed in [Teller88], in which it is suggested that each memory module maintain a table of resident pages. These tables are accessed associatively by virtual address; a miss indicates a page fault. Subsequent work has verified the performance advantages of translating virtual addresses to physical addresses at the memory rather than at the processor ([Teller94], [Qui98], [Qui01]).

A related idea is *inverted page tables* ([Houdek81], [Chang88], [Lee89]) which also features a one to one correspondence between page table entries and physical pages. However, the intention of inverted page tables is simply to support large address spaces without devoting massive amounts of memory to traditional forward-mapped page tables. The page tables still reside in memory, and translation is still performed at the processor. A hash table is used to locate page table entries from virtual addresses. In [Huck93], this hash table is combined with the inverted page table to form a *hashed page table.*

6.4 Multithreading

Multithreading is a very well known technique. In [Agarwal92] and [Thekkath94] it is shown that hardware multithreading can significantly improve processor utilization. A large number of designs have been proposed and/or implemented which incorporate hardware multithreading; examples include HEP [Smith81], Horizon [Thistle88], MASA [Halstead88], Tera [Alverson90], April [Agarwal95], and the M-Machine [Dally94]. Most of these designs are capable of executing instructions from a different thread on every cycle, allowing even single-cycle pipeline bubbles in one thread to be filled by instructions from another. An extreme model of multithreading, variously proposed as processor coupling [Keckler92], parallel multithreading [Hirata92] and simultaneous multithreading [Tullsen95], allows multiple threads to issue instructions during the *same* cycle in a superscalar architecture. To our knowledge, this idea has not yet been implemented in an actual processor.

The idea of using multithreading to handle events is also not new. It is described in both [Keckler99] and [Zilles99], and has been implemented in the M-Machine [Dally94]. Using a separate thread to handle events has been found to provide significant speedups. In [Keckler99] these speedups are attributed to three primary factors:

1. No instructions from the faulting thread need to be squashed
2. The thread's context does not need to be saved and subsequently restored
3. The faulting thread may continue to execute concurrently with the event handler

7 Timeline

An approximate timeline is useful for quantifying the amount by which we are behind schedule at any given point. Our chronological goals for completing the research outlined in this proposal are as follows:

Date	Milestone
June, 2001	Simulator completed
September, 2001	Benchmarks and Operating System completed
October, 2001	Area estimates via. layout completed
November, 2001	Evaluation of Sim completed
December, 2001	Evaluation of hardware LRU/page tables completed
January, 2002	Evaluation of capabilities, squids completed
February, 2002	Evaluation of sparse objects, trap bits completed
March, 2002	Evaluation of VLIW, multithreading completed
August, 2002	Thesis completed

References

[Abramson86] D. A. Abramson, J. Rosenberg, "The Micro-Architecture of a Capability-Based Computer", Proc. Micro '86, pp. 138-145.

[Agarwal92] Anant Agarwal, "Performance Tradeoffs in Multithreaded Processors", IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 5, September 1992, pp. 525-539.

[Agarwal95] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, Donald Yeung, "The MIT Alewife Machine: Architecture and Performance", Proc. ISCA '95, pp. 2-13.

[Ahuja95] Pritpal S. Ahuja, Douglas W. Clark, Anne Rogers, "The Performance Impact of Incomplete Bypassing in Processor Pipelines", Proc. Micro '95, pp. 36-45.

[Alverson90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, Burton Smith, "The Tera Computer System", Proc. 1990 International Conference on Supercomputing, pp. 1-6.

[Anderson86] M. Anderson, R. D. Pose, C. S. Wallace, "A Password-Capability System", The Computer Journal, Vol. 29, No. 1, 1986, pp. 1-8.

[Brown99] Jeremy Brown, "Memory Management on a Massively Parallel Capability Architecture", Ph.D. thesis proposal, M.I.T., December 1999.

[Carter94] Nicholas P. Carter, Stephen W. Keckler, William J. Dally, "Hardware Support for Fast Capability-based Addressing", Proc. ASPLOS VI, 1994, pp. 319-327.

[Case99] Brian Case, "Sun Makes MAJC With Mirrors", Microprocessor Report, October 25, 1999, pp. 18-21.

[Chang88] Albert Chang, Mark F. Mergen, "801 Storage: Architecture and Programming", ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988, pp. 28-50.

[Chase94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, Edward D. Lazowska, "Sharing and Protection in a Single-Address-Space Operating System", ACM Transactions on Computer Systems, Vol. 12, No. 4, November 1994, pp. 271-307.

[Chen92] J. Bradley Chen, Anita Borg, Norman P. Jouppi, "A Simulation Based Study of TLB Performance", Proc. ISCA '92, pp. 114-123.

[Chong95] Yong-Kim Chong, Kai Hwang, "Performance Analysis of Four Memory Consistency Models for Multithreaded Multiprocessors", IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 10, October 1995, pp. 1085-1099.

[Conte93] Thomas M. Conte, William Mangione-Smith, "Determining Cost-Effective Multiple Issue Processor Designs", Proc. ICCD '93, pp. 94-101.

[Dally92] William J. Dally, J. A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, Gregory A. Fyler, "The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms", IEEE Micro, April 1992, pp. 23-38.

[Dally94] William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, Whay S. Lee, "M-Machine Architecture v1.0", MIT Concurrent VLSI Architecture Memo 58, Massachusetts Institute of Technology, August 24, 1994.

[Dally98] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, Richard Lethin, Michael Noakes, Peter Nuth, Ellen Spertus, Deborah Wallach, D. Scott Wills, Andrew Chang John Keen, "The J-Machine: A Retrospective", 25 years of the international symposia on Computer architecture (selected papers), 1998, Pages 54 – 5.

[Denneau00] Monty Denneau, personal communication, Nov. 17, 2000.

[Dennis65] Jack B. Dennis, Earl C. Van Horn, "Programming Semantics for Multiprogrammed Computations", Communications of the ACM, Vol. 9, No. 3, March 1966, pp. 143-155.

[Diefen99] Keith Diefendorff, "Power4 Focuses on Memory Bandwidth", Microprocessor Report, October 6, 1999, pp. 11-17.

[Fabry74] R. S. Fabry, "Capability-Based Addressing", Communications of the ACM, Volume 17, Number 7, July 1974, pp. 403-412.

[Fillo95] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, Whay S. Lee, "The M-Machine Multicomputer", Proc. MICRO-28, 1995, pp. 146-156.

[Gaddis96] N. B. Gaddis, J. R. Butler, A. Kumar, W. J. Queen, "A 56-Entry Instruction Reorder Buffer", 1996 Digest of Technical Papers, ISSCC 1996, pp. 212-213.

[Gharach90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, John Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", Proc. ISCA '90, pp. 15-26.

[Gharach91] Kourosh Gharachorloo, Anoop Gupta, John Hennessy, "Performance Evaluation of Memory consistency Models for Shared-Memory Multiprocesors", Proc. ASPLOS '91, pp. 245-257

[Glaskow99] Peter N. Glaskowsky, "MoSys Explains 1T-SRAM Technology", Microprocessor Report, Vol. 13, No. 12, September 13, 1999.

[Gokhale95] Maya Gokhale, Bill Holmes, Ken Iobst, "Processing in Memory: The Terasys Massively Parallel PIM Array", IEEE Computer, April 1995, pp. 23-31.

[Greenblatt74] Richard Greenblatt, "The LISP Machine", Working Paper 79, M.I.T. Artificial Intelligence Laboratory, November 1974.

[Grossman98] J.P. Grossman, "Point Sample Rendering", Proc. 9th Eurographics Workshop on Rendering, June, 1998, pp. 181-192.

[Grossman00] J.P. Grossman, "Cheap Out-of-Order Execution using Delayed Issue", Proc. ICCD '00, pp. 549-551.

[Halstead88] Robert H. Halstead Jr., Tetsuya Fujita, "MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing", Proc. ISCA '88, pp. 443-451.

[Herbert79] A. J. Herbert, "A Hardware-Supported Protection Architecture", Proc. Operating Systems: Theory and Practice, North-Holland, Amsterdam, Netherlands, 1979, pp. 293-306.

[Hily99] Sébastien Hily, André Seznec, "Out-of-Order Execution May Not Be Cost-Effective on Processors Featuring Simultaneous Multithreading", Proc. HPCA '99, pp. 64-67.

[Hirata92] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, Teiji Nishizawa, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads", Proc. ISCA '92, pp. 136-145.

[Houdek81] M. E. Houdek, F. G. Soltis, R. L. Hoffman, "IBM System/38 Support for Capability-Based Addressing", Proc. ISCA '81, pp. 341-348.

[Huang01] Andrew Huang, "The Q-Machine", Ph.D. thesis proposal, MIT, April 2001.

[Huck93] Jerry Huck, Jim Hays, "Architectural Support for Translation Table Management in Large Address Space Machines", Proc. ISCA '93, pp. 39-50.

[IBM00] IBM, "Blue Gene Architecture", IBM press release, available at http://www.research.ibm.com/news/detail/architecture_fact.html

[Jacob97] Bruce Jacob, Trevor Mudge, "Software-Managed Address Translation", proc. HPCA '97, pp. 156-167.

[Keckler92] Stephen W. Keckler, William J. Dally, "Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism", Proc. ISCA '92, pp. 202-213.

[Keckler99] Stephen W. Keckler, Whay S. Lee, "Concurrent Event Handling through Multithreading", IEEE Transactions on Computers, Vol. 48, No. 9, September 1999, pp. 903-916.

[Knight00] Tom Knight, "Faster Water!", personal communication, 2000.

[Kowarik90] Oskar Kowarik, Rainer Kraus, Kurt Hoffman, Karl H. Horninger, "Associative and Data Processing Mbit-DRAM", Proc. ICCD '90, pp. 421-424.

[Kozyrakis97] Christoforos E. Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanović, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin gribstad, Kimberly Keeton, Randi Thomas, Noah Treuhaft, Katherine Yelick, "Scalabe Processors in the Billion-Transistor Era: IRAM", IEEE Computer, September 1997, pp. 75-78.

[Kuskin94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, John Hennessy, "The Stanford FLASH Multiprocessor", Proc. ISCA '94, pp. 302-313.

[Laudon97] James Laudon, Daniel Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server", Proc. ISCA '97, pp. 241-251.

[Lee89] Ruby Lee, "Precision Architecture", IEEE Computer, January 1989, pp. 78-91.

[Lee01] Jaejin Lee, Yan Solihin, Josep Torrellas, "Automatically Mapping Code on an Intelligent Memory Architecture", Proc. HPCA '01, pp. 121-132.

[Lenoski92] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, John Hennessy, "The DASH Prototype: Implementation and Performance", Proc. ISCA '92, pp. 92-103.

[Luk99] Chi-Keung Luk, Todd C. Mowry, "Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation", Proc. ISCA '99, pp. 88-99.

[Mai00] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, Mark Horowitz, "Smart Memories: A Modular Reconfigurable Architecture", Proc. ISCA '00, pp. 161-171.

[Margolus96] Norman Margolus, "CAM-8: A Computer Architecture Based on Cellular Automata", In A. Lawniczak and R. Kapral editors, *Pattern Formation and Lattice-Gas Automata*, American Mathematical Society, 1996, pp. 167-187.

[Margolus00] Norman Margolus, "An Embedded DRAM Architecture for Large-Scale Spatial-Lattice Computations", Proc. ISCA '00, pp. 149-160.

[Moon85] David A. Moon, "Architecture of the Symbolics 3600", Proc. ISCA '85, pp. 76-83, 1985.

[Nagle93] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, Richard Brown, "Design Tradeoffs for Software-Managed TLBs", Proc. ISCA '93, pp. 27-38.

[Oskin98] Mark Oskin, Frederic T. Chong, Timothy Sherwood, "Active Pages: A Computation Model for Intelligent Memory", Proc. ISCA '98, pp. 192-203.

[Oskin99a] Mark Oskin, Frederic T. Chong, Timothy Sherwood, "ActiveOS: Virtualizing Intelligent Memory", Proc. ICCD '99, pp. 202-208.

[Oskin99b] Mark Oskin, Justin Jensley, Diana Keen, Frederic T. Chong, Matthew Farrens, Aneet Chopra, "Exploiting ILP in Page-Based Intelligent Memory", Proc. MICRO '99, pp. 208-218.

[Palacharla97] Subbarao Palacharla, Norman P. Jouppi, J. E. Smith, "Complexity-Effective Superscalar Processors", Proc. ISCA '97, pp. 206-218.

[Pfister00] Hanspeter Pfister, Matthias Zwicker, Jeroen Van Baar, Markus Gross, "Surfels: Surface Elements as Rendering Primitives", Proc. SIGGRAPH '00, pp. 335-342.

[Pose01] Ronald Pose, "Password-Capabilities: Their Evolution from the Password-Capability System into Walnut and Beyond", proc. 6th Australasian computer Systems Architecture Conference, 2001, pp. 105-113.

[Qiu98] Xiaogang Qiu, Michel Dubois, "Options for Dynamic Address Translation in COMAs", Proc. ISCA '98, pp. 214-225.

[Qui01] Xiaogang Qiu, Michel Dubois, "Towards Virtually-Addressed Memory Hierarchies", Proc. HPCA '01, pp. 51-62.

[Rusink00] Szymon Rusinkiewicz, Marc Leroy, "QSplat: A Multiresolution Point Sample Rendering System for Large Meshes", Proc. SIGGRAPH '00, pp. 343-352.

[Saulsbury00] Ashley Salsbury, Fredrik Dahlgren, Per Strenström, "Recency-Based TLB Preloading", Proc. ISCA '00, pp. 117-127.

[Scott96] Steven L. Scott, "Synchronization and Communication in the T3E Multiprocessor", Proc. ASPLOS VII, 1996, pp. 26-36.

[SGI] SGI, "Performance of the Cray T3E™ Multiprocessor", SGI technical white paper, available at http://www.sgi.com/t3e/performance.html

[Smith81] Burton J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System, Proc. Real-Time Signal Processing IV, SPIE Vol. 298, 1981, pp. 241-248.

[Smith82] James E. Smith, "Decoupled Access/Execute Computer Architectures", Proc. ISCA '82, pp. 112-119.

[Soundar92] Vijayaraghavan Soundararajan, Anant Agarwal, "Dribbling Registers: A Mechanism for Reducing Context Switch Latency in Large-Scale Multiprocessors", Laboratory for Computer Science technical memo LCS TM-474, Massachusetts Institute of Technology, November 6, 1992.

[Talluri92] Madhusudhan Talluri, Shing Kong, Mark D. Hill, David A. Patterson, "Tradeoffs in Supporting Two Page Sizes", Proc. ISCA '92, pp. 415-424.

[Teller88] Patricia J. Teller, Richard Kenner, Marc Snir, "TLB Consistency on Highly-Parallel Shared-Memory Multiprocessors", Proc. 21st Annual Hawaii International Conference on System Sciences, 1988, pp. 184-193.

[Teller90] Patricia J. Teller, "Translation-Lookaside Buffer Consistency", IEEE Computer, Vol. 23, Iss. 6, June 1990, pp. 26-36.

[Teller94] Patricia J. Teller, Allan Gottleib, "Locating Multiprocessor TLBs at Memory", Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences, 1994, pp. 554-563.

[Thekkath94] Radhika Thekkath, Susan J. Eggers, "The Effectiveness of Multiple Hardware Contexts", Proc. ASPLOS VI, 1994, pp. 328-337.

[Thistle88] Mark R. Thistle, “A Processor Architecture for Horizon”, Proc. Supercomputing ’88, pp. 35-41.

[Tullsen95] Dean M. Tullsen, Susan J. Eggers, Henry M. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism”, Proc. ISCA ’95, pp. 392-403.

[Tyner81] P. Tyner, “iAXP 432 General Data Processor Architecture Reference Manual”, Intel Corporation, Aloha, OR, 1981.

[Waingold97] Elliot Waingold, Michael Taylor, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Srikrishna Devabhaktuni, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, Anant Agarwal, “Baring it all to Software: The Raw Machine”, IEEE Computer, Vol. 30, Iss. 9, Sept. 1997, pp. 86-93.

[Ward88] Stephen A. Ward, Robert C. Zak, “Set-Associative Dynamic Random Access Memory”, Proc. ICCD ’88, pp. 478-483.

[Zilles99] Craig B. Zilles, Joel S. Emer, Gurindar S. Sohi, “The Use of Multithreading for Exception Handling”, Proc. Micro ’99, pp. 219-229.

[Zucker92] Richard N. Zucker, Jean-Loup Baer, “A Performance Study of Memory Consistency Models”, Proc. ISCA ’92, pp. 2-12.