♈-12 February 12, 2001

Hamal Design Rationale

J.P. Grossman

November 14, 2000 – February 12, 2001

1 Introduction

The Hamal Instruction Set Architecture was developed over a period of more than a year by a process which can fairly be described as the worst sort of engineering. It is almost purely the result of thought experiments and hypothetical debates. With the exception of some preliminary assembly programming and scattered ties to existing architectures, it has benefited from little to no real-world validation. Nonetheless, many aspects of the design process have been educational. Countless arguments at meetings and on whiteboards have provided insight into many of the subtle challenges of system design. An unusually diverse set of backgrounds and philosophies within the Aries group have given rise to a number of novel macro- and micro-architectural mechanisms which support a wide range of computational paradigms. The resulting architecture may, at first, seem arbitrary, but in fact every detail is the result of careful thought and difficult compromises.

The purpose of this design rationale is to document the thought processes and arguments which gave rise to the Hamal ISA, as well as some of the design alternatives which were considered but rejected. The material contained herein represents countless heated discussions which must at times have resembled a bad day of the middle east peace negotiations. Despite outward appearances, however, these debates have been both stimulating and rewarding to all parties involved, and I have enjoyed working with a group of intelligent and competent researchers whose philosophies differ so markedly from my own.

2 Design Principles

A number of general principles have guided the design of the Hamal architecture. They are presented below roughly in order from most important to least important. While it is difficult to raise issue with this set of principles, the order of importance is my own and does not necessarily reflect the opinions of other group members.

Project Aries Technical Memo ARIES-TM-12
Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA

2.1 General Purpose Parallel Architecture

One of the primary goals of the Hamal effort is to produce a truly general purpose parallel architecture in the sense that it supports not only a wide range of applications, but also a spectrum of programming languages and environments. This is a daunting task and the resulting architecture will inevitably be suboptimal for any specific application. However, it has the advantage of forcing the designers to consider computation in a broad sense, and it helps us to avoid the temptation of migrating platform-specific software mechanisms into hardware. The only restriction on the applications of interest is that they be *parallel* applications; the gentlemen at AMD have proven to be quite adept at developing hardware support for scalar applications, and we are happy to leave this task in their able hands.

2.2 Silicon Efficiency

Silicon efficiency, roughly defined as performance per unit area, is an important factor in determining both the cost and overall performance of a scalable multiprocessor. The Hamal architecture has been designed to maximize silicon efficiency. This design philosophy favours small changes in hardware which produce significant gains in performance, while eschewing complicated features with large area costs. It also favours general mechanisms over application or programming language specific enhancements.

The difficulty with silicon efficiency as a metric is that it is extremely application-dependent. Applications differ wildly in terms of their computational intensity, memory usage, communication requirements, parallelism and scalability. It is not possible to maximize silicon efficiency in an absolute sense without reference to a specific set of applications, but one can often argue convincingly for or against specific architectural features based on this design principle.

2.3 Simplicity

In short, simpler is better. Simplicity is often a direct consequence of silicon efficiency, as many complicated mechanisms improve performance only at the cost of overall efficiency (e.g. out of order issue, branch prediction, register renaming, speculative execution, to name a few). Simplicity also has advantages that silicon efficiency on its own does not; simpler architectures are faster to design, easier to test, less prone to errors and friendlier to compilers.

2.4 Programmability

In order to be useful, an architecture must be easy to program. This means two things: it must be easy to *write* programs, and it must be easy to *debug* programs. To a large extent, the former requirement can be dealt with by the compiler so long as the underlying architecture is not so obscure as to defy compilation. The latter requirement can be partially addressed by the programming environment, but there are a number of hardware mechanisms which can greatly ease and/or accelerate the process of debugging. It is perhaps more accurate to refer to this design principle as "debuggability" rather than "programmability", but on the other hand one can argue that there is no difference between the two; it has been said that "programming consists of debugging a blank piece of paper".

2.5 Scalability

In the Aries group we have a word that we like to use to describe computers with fewer than a thousand processors – "scalar". Traditional approaches to parallelism do not scale very well beyond a thousand nodes, in part due to the need to maintain large amounts of globally coherent state at each node. The Hamal architecture has been designed to overcome this barrier and scale to a million or even a billion nodes.

2.6 Performance

Last and least of the design principles is performance. Along with simplicity, performance can to a large extent be considered a subheading of silicon efficiency. They are opposite subheadings; the goal of silicon efficiency gives rise to a constant struggle between simplicity and performance. By placing performance last among design principles I do not intend to imply that it is unimportant; indeed my interest in Hamal is above all else to produce a terrifyingly fast machine. Rather, I am emphasizing that a fast machine is uninteresting unless it supports a variety of applications, it is economical in its use of silicon, it is practical to build and program, and it will scale gracefully over the years as the number of processors is increased by multiple orders of magnitude.

3 Design Overview

Hamal is a distributed, shared-memory machine. "distributed" hardly needs justification; anyone who believes in uniform memory access for a massively parallel machine has their head in the sand. Hamal was designed with the vision of unifying processor and memory. The basic design element is a small processor-memory node which is replicated across the system and connected by a high-performance network. The amount of memory at the nodes is relatively small; as such supporting transparent shared-memory is a must.

3.1 Capabilities

Hamal is a capability architecture; all memory references are performed using self-contained 128 bit capabilities. This allows the hardware to guarantee that user programs will make no illegal memory references without requiring any form of capability/segment table. It is therefore safe to use a single shared virtual address space which greatly simplifies the memory model. Additionally, the number of segments is essentially unbounded; in particular object-based protection schemes become practical.

The advantages of capabilities must, of course, be weighed against the costs of doubling the pointer size. Not only does this require increased storage for capabilities in memory, but it also doubles the size of the general purpose registers and at least some datapaths. We feel that these costs are more than justified by the advantages outlined above.

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. This 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, 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.

3.3 Distributed Objects

In any large-scale system, the layout of data in physical memory is crucial to achieving the maximum possible performance. In particular, many good parallel algorithms require data objects to be distributed across multiple nodes in the system. Providing support for such distributed objects is non-trivial. For example, the naïve approach of using a fixed set of virtual address bits to identify a physical node has the significant disadvantage that distributing an object across multiple nodes requires allocating many non-contiguous blocks of the virtual address space for the single object, complicating both memory management and object references.

Some time ago I proposed *multi-striped addressing* to allow contiguous portions of the virtual address space to be striped across nodes with any power-of-two granularity. The central idea was to use the top five virtual address bits to index the node ID within the virtual address (Figure 1). This idea is simple and elegant, but has two significant drawbacks. First, it restricts the size a segment's *facets* (portions of the segment which exist on various nodes) to be powers of two in size. Second, it can introduce large amounts internal fragmentation of physical pages.

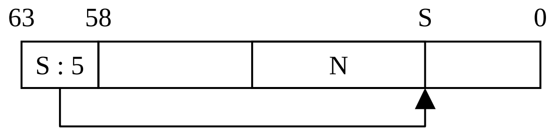

Figure 1: Multistriped Addressing

An alternative approach suggested by Jeremy Brown is *sparse objects*. In this scheme, the upper virtual address bits are used to identify a physical node. When a segment of any size is allocated, it is implicitly allocated on *all* nodes. Individual facets, however, are not allocated until they are used. The key mechanism of sparse objects is a translation table which exists at the boundary of each processing node to translate local virtual addresses to/from global segment unique identifiers (GSUID) and an offset.

When a pointer moves from a node to the network, it is first decomposed into a segment base address and an offset. The base address is then 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 pointer to a sparse object leaves its home node, a new GSUID is created which consists of the node identifier and 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 its facet of a distributed object, then the facet is allocated and the base address is entered into the table.

The sparse object scheme has the large advantage that all memory on a given node is allocated locally. In particular, this means that it can be allocated contiguously, which almost completely eliminates internal fragmentation of pages (there may still be some fragmentation due to alignment restrictions of allocated segments). It has the additional advantages of allowing arbitrarily sized facets, and enabling local compacting garbage collecting to proceed completely transparently to the rest of the system. The scheme also has the significant disadvantage of requiring complicated translation caches as well as kernel software to handle translation cache misses. Our initial feeling is that the benefits of this scheme outweigh its costs; simulation will provide us with a more accurate evaluation.

4 Processor Design

The Hamal architecture features 128 bit multi-context Very Long Instruction Word (VLIW) processors with support for predicated execution. Context 0 is reserved for event handling; contexts 1-3 are “user contexts” which run user programs. The following sections will attempt to justify these specifications given our working set of design principles.

4.1 Datapath Size

The choice of 128 bits as the basic datapath size is motivated by two factors:

1. Capabilities are 128 bits, so at least some datapaths must be this wide
2. Wide datapaths make effective use of the available embedded DRAM bandwidth

A criticism of wide datapaths is that large portions of the register file and/or functional units will be unused for applications which deal primarily with 32 or 64 bit data, significantly reducing the area efficiency of the processor. This issue is addressed in two ways. First, each register is addressable as a single 128 bit register, two 64 bit registers, or four 32 bit registers. This requires a small amount of shifting logic to implement in hardware, and increases both the register file utilization and the number of registers available to user programs. Second, many of the instructions can operate in parallel on two sets of 64 bit inputs or four sets of 32 bit inputs, packed into 128 bit registers. This provides the opportunity to increase both performance and functional unit usage via fine-grained SIMD parallelism. While intuitively appealing, it is difficult to properly evaluate these SIMD instructions at such an early stage. Further simulation and design work is required to obtain estimates of their performance advantages and hardware costs.

4.2 Hardware Multithreading

A single thread of execution typically cannot issue instructions on every cycle due to cache misses and branch mispredictions. This problem is compounded in architectures such as Hamal which do not provide branch prediction or data caches. Hardware multithreading can recover many of these lost cycles and improve silicon efficiency by allowing different threads to issue instructions on consecutive cycles, thus allowing bubbles in one thread to be filled by another. An additional potential benefit is the elimination of the context switch overhead normally required to service asynchronous interrupts. If a hardware context is reserved for handling events, then no state needs to be saved when an interrupt occurs or restored when the handler exits, and both context switches occur in a single cycle.

The benefits of hardware multithreading are compelling, but must be weighed against two significant costs:

1. Cache and/or local memory must be shared between a number of different threads
2. Registers must be duplicated for each hardware context

There is no good solution to the first problem. A dangerous temptation is to increase the size of the cache/local memory to compensate for the extra threads. However, to do this is to forget that the goal of multithreading is to provide improved silicon efficiency. If both the registers and memory are to be duplicated, then one might as well improve performance significantly at a small additional cost by using multiple single-threaded processors rather than a single multithreaded one. A hardware designer completely blind to reality might then propose the next logical step which is to add some additional registers to each of these processors to make them multithreaded… lather, rinse, repeat.

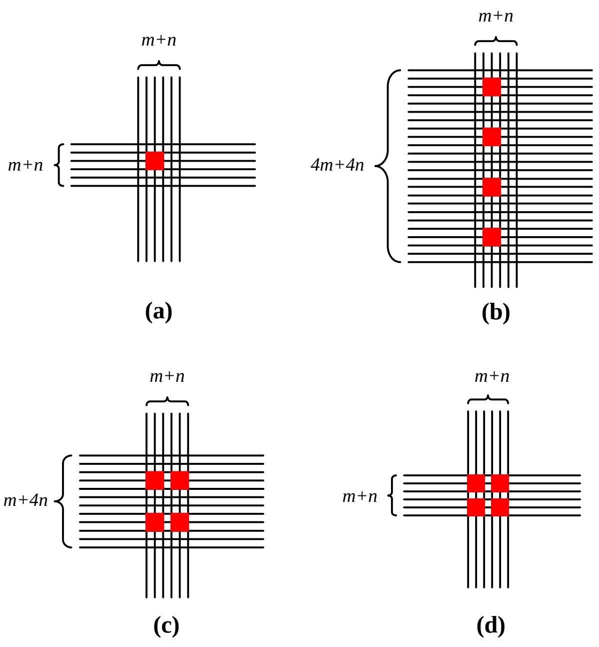

Figure 2: Read and write ports

The second cost is not quite as large as it seems. Although the registers must be duplicated, the dominant cost in a multi-ported register file is not registers but wires. A register file with *m* read ports and n write ports requires $m+n$ control lines per register and $m+n$ bit lines per bit (Figure 2a). A naïve approach to duplicating the register file k times is to simply duplicate all the control lines, resulting in a register file k times as large (Figure 2b). However, if only one thread can issue instructions on a given cycle then only one thread can read from the register file on a given cycle. Hence, m read control lines can be shared by k registers, so the number of control lines required is only $m+kn$ per register (Figure 2c). Typically $m \approx 2n$, so this reduces the number of control lines from $3kn$ to $(2+k)n$. For $k = 4$, this represents a factor of two savings in area.

We could further reduce the area overhead by imposing the restriction that on a given cycle at most one thread can read or write the register file. This allows both read and write control lines to be shared by k registers, eliminating most area overhead (Figure 2d). However, this restriction implies that all pipelines must be drained before performing a context switch, since otherwise instructions from one thread may complete and need to write to the register file on the same cycle that another thread is reading from the register file. Consequently, much of the advantage of hardware multithreading is lost since the original motivation was to be able to fill small (say 1-5 cycle) pipeline bubbles.

In the Hamal architecture, context 0 is reserved for handling asynchronous events. As mentioned previously, this allows such events to be attended to without any context switch overhead. It could be argued that if asynchronous events are infrequent, then this is a waste of a context which could be used for user computation. On the other hand, extra contexts are only useful up to a certain point, namely the point at which there are enough contexts to hide latencies and keep the hardware busy at all times. Once this number of contexts has been reached, one can safely add an event handling context without being tempted to make it general purpose.

4.3 VLIW

Very Long Instruction Word architectures allow instruction level parallelism to be explicitly scheduled by the compiler. For many applications they provide the same performance advantages as dynamic superscalar processors, and they do so at a fraction of the hardware cost and complexity. As such, VLIW is an appealing way to improve silicon efficiency.

While a VLIW design is significantly simpler than dynamic superscalar, it is not without cost. The two main costs of VLIW (there seem to be two of everything in this document) are register file ports and instruction word size. Note that functional units are not an implicit cost of VLIW design; a VLIW architecture need not have any more basic arithmetic units than a scalar architecture. The difference is that the VLIW architecture allows multiple units to be used on a single cycle.

The first cost, register ports, is significant. Each operation in a very long instruction must be able to read its source operands at issue time and write a result at commit time. This places restrictions on the width of the instruction word and/or the connectivity of the register file. For truly long instruction words containing a large number of operations (the Multiflow VLIW machine was designed for up to 28 operations per instruction word), it is necessary to partition the register file among functional units. This makes life difficult for the compiler, not only because of loss of orthogonality and the painful scheduling problem that this creates, but because advanced compilation techniques such as trace scheduling are required to make effective use of so many functional units. Thus, in the interest of simplicity and programmability, the Hamal architecture will only support three instructions per long instruction word (one arithmetic instruction, one memory instruction and one control instruction).

The second cost, instruction word size, is difficult to evaluate. Because many instruction groups will contain empty slots, more instruction memory will be required to hold a VLIW program. This cost must be weighed against the performance advantages of VLIW to determine whether or not overall area efficiency is improved. The difficulty is that this is extremely application-dependent. Programs with a high degree of instruction-level parallelism (ILP) will make good use of the VLIW instruction slots, resulting in fast, compact code. Programs with less ILP, on the other hand, will contain more empty slots and will be unable to exploit the potential performance advantages of VLIW. It is our hope that most interesting parallel applications will contain enough ILP to justify a VLIW architecture. This is one of the many aspects of the Hamal architecture which will require simulation to properly evaluate.

4.4 Predicated Execution

Conditional branches are not "happy" instructions. In a vanilla architecture, the fetch mechanism must wait for the condition to be evaluated to determine the address of the next instruction. This introduces bubbles into the pipeline; the deeper the pipeline, the bigger the hurt. Modern architectures address this problem with branch prediction and speculative execution. This technique is effective (branch prediction accuracy in current architectures is around 95%), but costly. Branch target buffers consume significant amounts of silicon area, and speculative execution increases the complexity of the design.

An alternate approach to conditional computation is the use of predicated execution. By allowing individual instructions to be predicated on the value of single bit predicates, many conditional branches can be eliminated (in particular, if-then-else blocks with short bodies can be implemented using predicated execution). This keeps the instruction stream linear and has the added benefit of enlarging basic blocks, which makes it easier for the compiler to optimize and schedule the instructions.

One important difference between conditional branches and predicated execution is that while a conditional branch freezes the instruction stream until the condition has been evaluated, predicated execution does not. A predicated instruction may issue freely and does not need to block for the predicate until it is ready to cause a side effect (e.g. register writeback). Predicated execution is in effect compiler controlled speculative execution. As such, it can be used to implement static branch prediction. Consider the following sequence of instructions:

```
        p1 = test r1, 1
(p1)    branch _somewhere
        r2a = fmul32 r7a, r7b
        r3 = load128 r5[0]
        ...
```

The branch instruction will hold up the pipeline waiting for p1 to become valid. We can avoid this performance penalty in the case that p1 evaluates to false by statically predicting that the branch will not be taken as follows:

```
        p1 = test r1, 1
(!p1)   r2a = fmul32 r7a, r7b
(!p1)   r3 = load128 r5[0]
(p1)    branch _somewhere
        ...
```

By migrating an appropriate number of instructions above the branch, the compiler can ensure that the predicate will be ready by the time the branch is encountered. Note that this only works for instructions that are not already predicated.

There is of course a cost associated with predicated execution. The hardware mechanisms are fairly inexpensive; more significant is the fact that a number of bits are added to each instruction (six in the Hamal architecture). This increases the size of the code by roughly 20%. Once again, we will rely on simulation experiments to determine whether or not the performance advantages of predicated execution are sufficient to justify this cost.

One subject of discussion has been whether to store predicates in the general purpose registers or a special purpose predicate register file. Using the general purpose registers has the advantage of simplifying the hardware design, but has the significant cost of adding three read ports to the registers. Additionally, for code that makes use of a large number of predicates, keeping single bit predicates in 32 bit registers is wasteful and reduces the number of general purpose registers available for computation. For these reasons, the predicates are stored in a separate predicate register file. Another suggestion has been to designate one of the general purpose registers as the predicate register file, but this actually complicates the hardware by destroying the uniform semantics of the general purpose registers.