Memory Integration: Implementing the Memory Model in Hardware

J.P. Grossman

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

http://www.ai.mit.edu

The Problem: 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. After several decades of computing, memory models have expanded to include mechanisms such as virtual memory, segmentation, and protection in a shared environment, yet the underlying hardware has not changed (other than becoming faster and denser). 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 segment descriptors and 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 segments/pages, but is of little use to programs whose working set of segments and/or pages is large. For example, an attempt to use segments to implement per-object protection would be disastrous for programs with a large working set of objects. Another problem common to any form of caching is the "pollution" that occurs in a multithreaded environment; a single TLB must be shared by all threads which reduces its effectiveness and introduces a cold-start effect at every context switch. Thus, reliance on a TLB is an obstacle to fine-grained multithreading. Finally, in a multiprocessor environment the TLB's must be kept globally consistent which places constraints on the scalability of the system.

Motivation: An alternate approach which is conceptually appealing is to introduce hardware mechanisms which directly implement the memory model. With the recent ability to place logic and memory on the same die, this approach may be not only appealing but also practical. There is an opportunity to explore novel memory system architectures with improved support for flexible security, efficienct multithreading and high scalability (beyond one million nodes).

Previous Work: A number of interesting mechanisms have been described in the literature and/or implemented in commercial machines. Guarded Pointers [2], introduced by Carter et. al., are a form of unforgeable capabilities [3] which include both a pointer and segment information within the guarded pointer itself. 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 address centrifuge in the Cray T3E [4] is used to distribute virtual addresses among physical processing nodes in a flexible and relatively efficient manner. For each index into a large distributed object, a mask specifies which bits are used to form the processing element number. This allows data to be arranged in a hypercube divided into identical sub-cubes, where all sides are arbitrary powers of two and individual sub-cubes are mapped to a single processing element. Furthermore, no global translation tables are required.

Both the Tera [1] and the Cray T3E [4] support simple atomic read-modify-write memory operations such as adding an integer to the contents of a memory location. These operations are useful for synchronization and are inexpensive to implement in hardware.

Approach: The Hamal architecture is a shared memory multiprocessor with a number of hardware mechanisms which directly implement the memory model. The architecture uses 129 bit capabilities which are broken down into 1 tag bit to distinguish pointers from data, 64 bits of virtual address, and 64 bits of segment information (permission bits, segment size, etc). Our capability format supports pointer arithmetic, nearly-tight object bounds (less than 6% internal fragmentation), sub-segmentation, exact array bounds, and forwarding pointers.

Virtual to physical address translation is performed by the memory itself rather than by 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. The mapping from virtual addresses to physical nodes is defined by taking the node ID from the upper bits of the virtual address. This is a fixed mapping and thus allows data to be located without storing any global information at each node.

Support for distributed data structures is provided via "sparse objects". When the operating system allocates a segment on a given node, it may set a bit in the capability to indicate that the object is sparse. This implicitly allocates the same segment on all nodes in the system. However, explicit allocation is performed lazily as other nodes in the system encounter the capability. When a sparse capability moves from a node to the network, the hardware uses a translation cache to convert it into a global segment unique identifier (GSUID) and an offset. When a GSUID arrives at a node, the hardware again uses the translation cache to convert it to a local address. Cache misses are handled by the operating system, allowing software to create the GSUID the first time a sparse capability leaves its home node and allocate segments on demand as new nodes encounter the capability.

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 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 similar to those of the Tera and Cray T3E memory systems [1, 4].

Impact: The memory model and its implementation have a direct impact on system performance, programmability and scalability. Capabilities provide a flexible security model and guarantee the safety of a shared virtual address space. A single virtual address space reduces the amount of state associated with a thread of execution, resulting in more efficient multithreading. Implementing virtual memory at the memory rather than at the processor, using a combination of hardware page tables and multistriped addressing, obviates the need for global translation tables or TLB's and therefore improves system scalability. Atomic memory operations augment existing datapaths with small amounts of logic to provide an efficient set of synchronization primitives.

Future Work: A cycle accurate C++ simulator for the Hamal architecture is currently under development. Running benchmark programs using various hardware configurations will allows us to quantify the costs and benefits of the described mechanisms. Other related work includes the design of an operating system to complement the hardware mechanisms provided by the Hamal memory system, and the development of programming language primitives which allow programmers to easily make use of these mechanisms.

Research Support: This work is supported by DARPA/AFOSR contract number F306029810172.

References:

[1] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Perterfield, and Burton Smith. The tera computer system. *Proc. 1990 ACM International Conference on Supercomputing*, pages 1–6, June 1990.

[2] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. *Proc. ASPLOS VI*, pages 319–327, Oct 1994.

[3] R. Fabry. Capability-based addressing. *Communications of the ACM, Vol. 17, No. 7*, pages 403–412, July 1974.

[4] Steven L. Scott. Synchronization and communication in the cray T3E multiprocessor. *Proc. ASPLOX VII*, pages 26–36, Oct 1996.