A Proposed Architecture for a Massive Memory Machine

by
Daniel Lopresti

Department of Computer Science
Princeton University

CS-TR-011-85

October 6, 1985
Revised October 30, 1985

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0456 and N00014-85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

A Proposed Architecture for a Massive Memory Machine

Daniel Lopresti

1 Introduction

A massive memory machine (M^3), as proposed in [1], is a supercomputer with enormous primary storage. As such, it promises tremendous performance improvements for programs that exhibit poor locality of reference and cause conventional computers to page-fault frequently, or "thrash." Applications from VLSI design, artificial intelligence, and database systems would benefit from this treatment. Further, such a machine may change fundamentally the way programs are written; in many applications space can be traded for time, but space has previously been a precious commodity. In this paper we propose an architecture for a massive memory machine.

2 A Hierarchical Massive Memory Machine

We illustrate this discussion with specific hardware. In particular, we assume that M^3 uses a 32 bit processor and that random access memory is packaged one megabyte to a board. The architecture to be presented generalizes to larger address spaces and denser memories.

A hierarchical M^3 is shown in figure 2.1. Each node in the tree is a nine slot card cage with a standard bus. Inner nodes are connected to a parent and eight children by high speed, point-to-point links. The lone processor, at the root, addresses four gigabytes of physical memory in the leaves through the four level hierarchy.

The time for a reference to pass through the tree is far less than that needed to service a page fault, hence this architecture will easily out-perform existing computers on the memory intensive applications suggested for M^3. An advantage of this arrangement over a conventional bus or the ESP scheme described in [1] is that as memory is increased the bus delay grows logarithmically, not linearly.

A disadvantage is that **every** memory access must traverse each level of the tree; a write requires one pass while a read takes two. For an instruction fetch, the most common memory reference a processor makes, the address must filter down through the hierarchy, then the opcode has to bubble back up. Since the delay between nodes is likely to be non-trivial, probably on the order of memory cycle time, this M^3 will

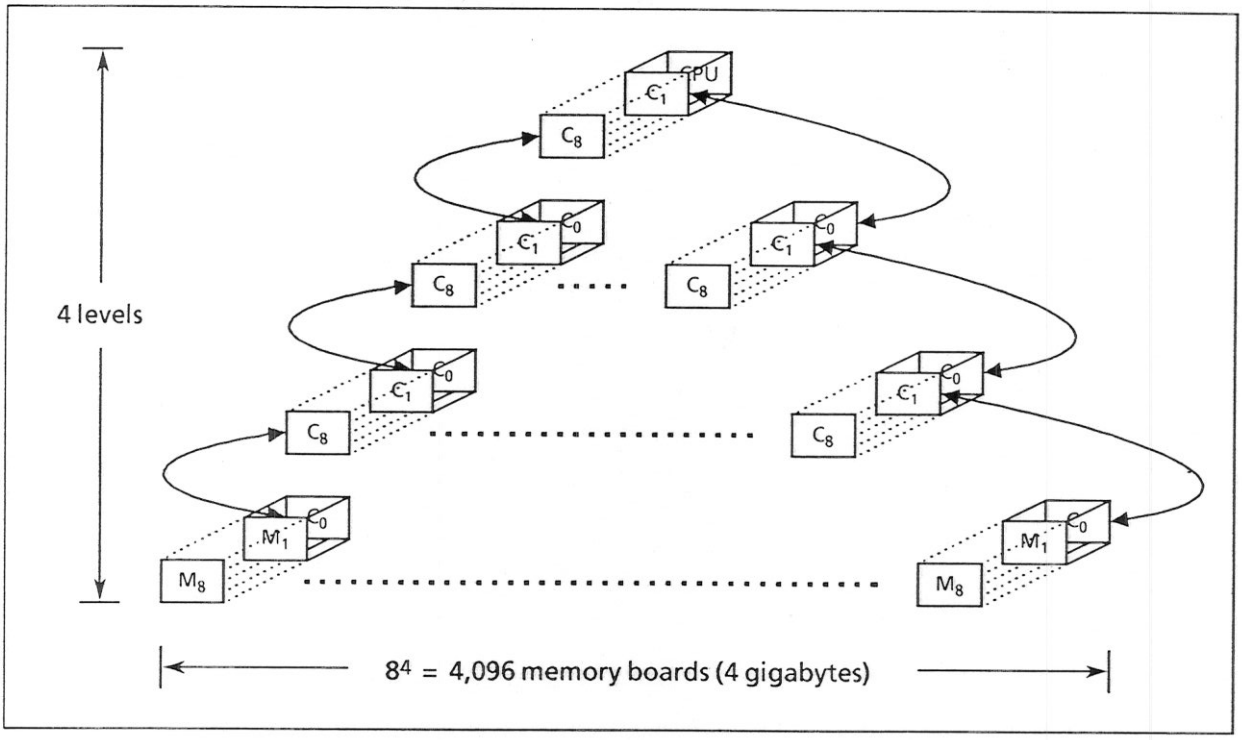

figure 2.1 A Hierarchical Massive Memory Machine

execute programs which have modest memory requirements much more slowly than a conventional computer would. In particular, this architecture's sluggish handling of operating system routines is a concern.

Although the custom hardware needed to realize this design is minimal, consisting only of the node-to-node connector boards and links, there are other important, open questions: where in the hierarchy are disk drives attached? How are programs loaded? Can multiprogramming be supported? Should the memory be fixed, or paged? This architecture is sufficiently different from existing machines that many answers are not obvious.

We now demonstrate how these issues can be resolved, while retaining the benefits of hierarchical massive memory.

3 A Hybrid Massive Memory Machine

We make some observations based on principles of existing operating systems and programming languages, in particular UNIX™ and C.

First, partition memory references into those made by "user" processes and those made by (operating) "system" routines. Further subdivide each of these categories into "text" (program instructions), "stack" (local variables), and "heap" (dynamic and global structures). We now suggest that, no matter what application is run on M3, the system code will be well-behaved. That is, system text, stack, and heap references will not be responsible for many page faults. This is certainly true with UNIX, which has a combined text, heap, and stack space of less than 300 kilobytes; it runs core resident at all times.

Further, we argue that the text space of programs written by humans is never very large and exhibits great spatial locality of reference; indeed, block-structured languages like C encourage this. Hence, instruction fetches rarely cause page faults. Also, by definition, access to a stack is restricted to its top element (although in practice a number of parameters and local variables physically near the top may be addressed). Thus stack operations are seldom responsible for page faults. The remaining segment, the user heap, contains a program's dynamic data structures as well as its global and static variables. VLSI design tools, expert systems, compilers, LISP interpreters, and database managers all use tremendous amounts of heap space. We venture that it is precisely these memory references that cause the thrashing which M3 is designed to avoid.

So here we pose the question, "why should well-behaved program segments be penalized because the user heap misbehaves?" The answer is that they need not be.

Consider the hybrid M3 shown in figure 3.1. This machine has two gigabytes of physical memory organized hierarchically as before; we call this memory the (user) "heap." The other two gigabytes of address space are virtual memory, mapped to four megabytes of RAM on the processor's own bus; this memory contains system code and user text and stack space. References to this local memory are just as fast as in a conventional computer. A memory map is depicted in figure 3.2.

This hybrid M3 has many advantages over the fully hierarchical model. Most importantly, a large percentage (certainly greater than 50%) of all memory references can be handled rapidly, on the processor's local bus. Also, there is an immediate cost savings as we do not waste physical memory on code segments that are known to run well in virtual memory. Finally, this architecture is much closer to existing machines than the hierarchical system, so many of the questions raised earlier ("Where in the hierarchy are disk drives attached?" "How are programs

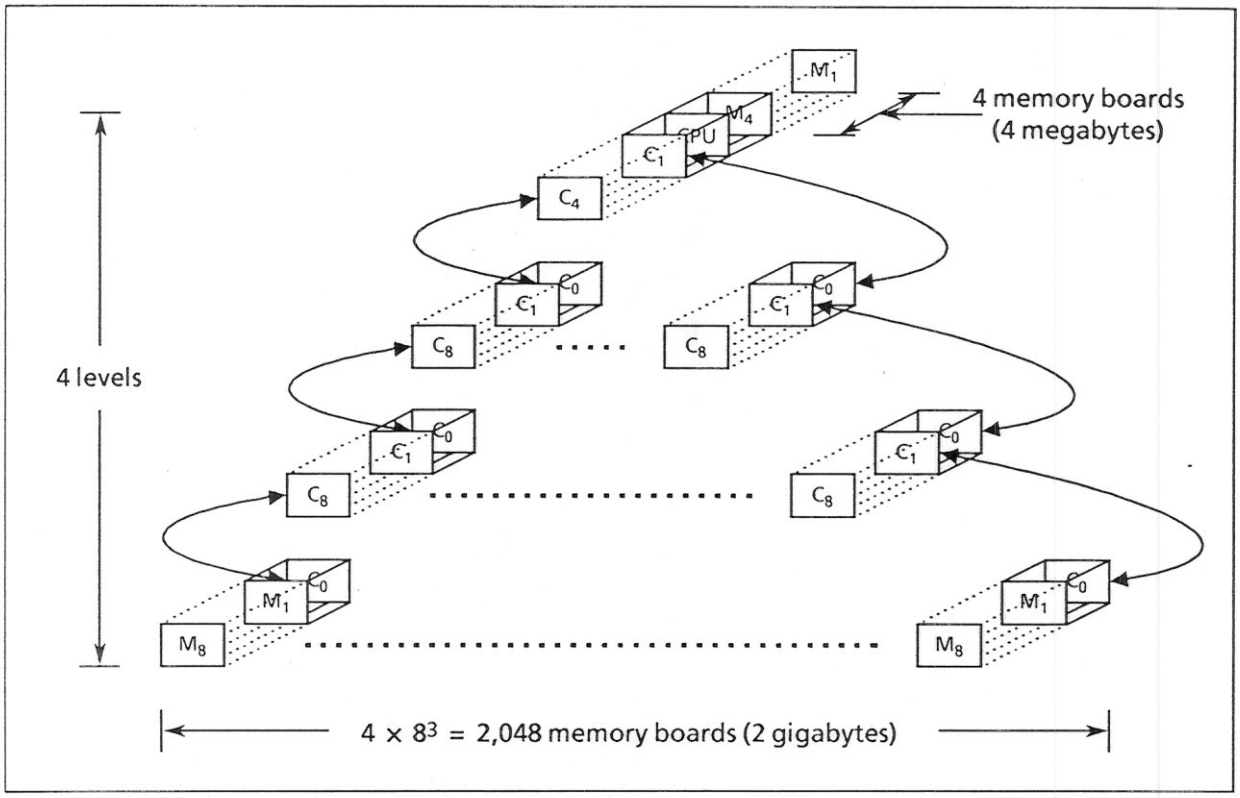

figure 3.1 A Hybrid Massive Memory Machine

loaded?" "Can multiprogramming be supported?") have already been answered. The difference is that the user heap is given a tremendous amount of physical memory to reside in. The tree, although slower than local memory, is a very fast alternative to storing large data structures on disk.

We emphasize two points. First, the local memory is not a cache, it is a portion of the virtual address space which can be paged. Although we encourage placing a cache close to the processor, M3 must not depend on a program exhibiting temporal locality. Second, the heap tree, while virtually mapped, is not paged. For a paging scheme to be effective requires spatial locality of user heap references, a property M3 applications definitely do not have.

An intriguing variation on this memory management scheme, one which would not require partitioning references, is to begin executing every program entirely from local memory. Then, have the operating system dynamically relocate segments of badly faulting processes to the heap tree instead of paging them out to disk.

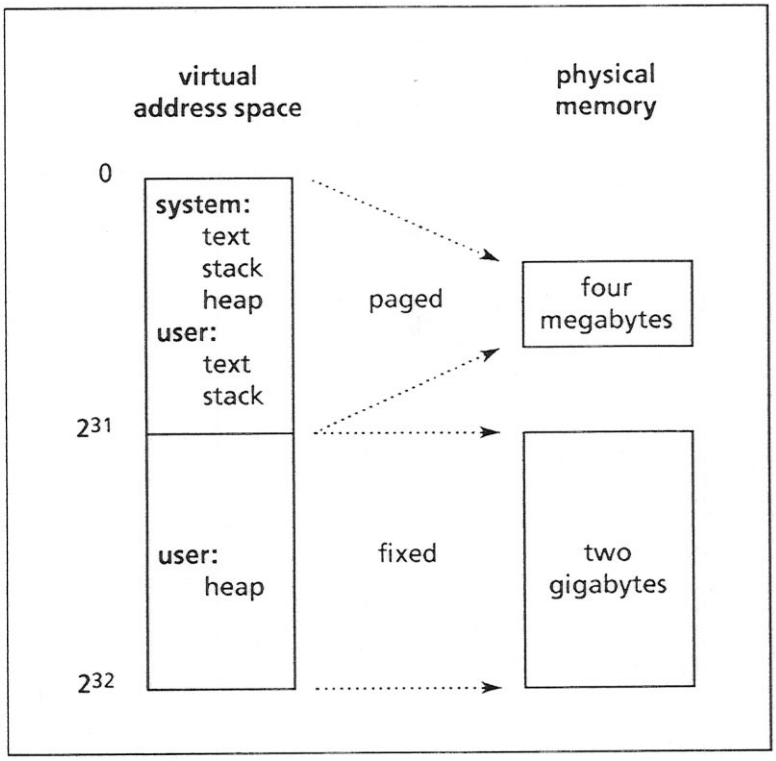

figure 3.2 A Memory Map

4 A Massive Memory Machine with Distributed Execution Units

We now offer a modification of the hybrid M3 which will, in some cases, speed up references to data in the heap tree. We are motivated by the observation that an instruction of the form:

$$INC\,(R0)$$

(an increment of the memory location pointed to by register R0) requires issuing the address contained in R0 down through the tree, bubbling the correct data value back up, incrementing it in the processor, and passing it back down again. This sequence is depicted in figure 4.1.

If we make the nodes more intelligent we can cut these three passes through the tree down to one. We simply provide each with an execution unit. Then, instead of moving the data to the processor, we move the processing to the data. More specifically, in the above case the processor issues the **instruction** down the tree. After decoding it, the appropriate node fetches the correct data value and performs the increment locally. This sequence is depicted in figure 4.2. Of course, this scheme

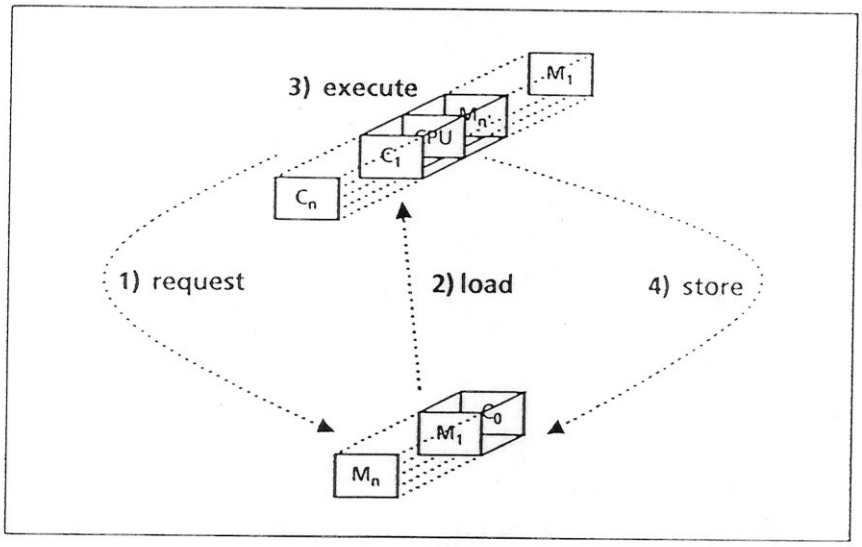

figure 4.1 Three Passes

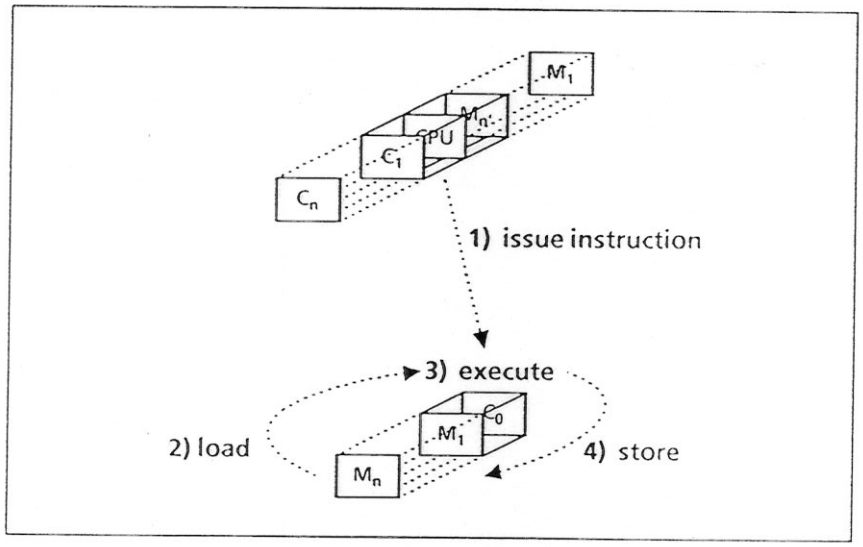

figure 4.2 One Pass

could be extended to include instructions which reference any number of operands anywhere in the address space, the updates being performed in the lowest common parent node. The time savings could be significant.

It may even be possible to exploit concurrency since the execution units are distributed. In particular, repetitive, time-consuming operations could be down-loaded to the lowest nodes and executed locally. Dynamic RAM refresh, memory initialization, and LISP garbage collection could all be handled in this way.

Unlike the simple hybrid architecture, this extension would require significant custom hardware. Most noticeably, the instruction-broadcasting processor would be very different from any CPU commercially available today.

Finally, we remark that this short cut will not work in every case; moving heap data into the processor becomes necessary when it directs the flow of instructions, as in a conditional branch.

5 Conclusions

We conclude by summarizing the assumptions underlying our proposed architecture. The hybrid massive memory machine is valid if user heap references are responsible, directly or indirectly, for most page faults. Execution units distributed throughout the nodes of the heap tree will improve performance whenever operations referencing the user heap values are simple updates.

Of necessity, this note ignores many practical issues, including fault tolerance, layout of the heap tree, packaging constraints, and dynamically managing the large physical memory. These, and other concerns, merit further study.

Lastly, yet to be determined is a name for this hybrid massive memory machine, one which yields a better acronym than "hmmm."

6 Acknowledgments

The author thanks Peter Honeyman and Steven Kugelmass for their comments on the first version of this paper.

7 References

[1] R. J. Lipton, et. al. "The Massive Memory Machine Project," Department of Electrical Engineering and Computer Science, Princeton University, 1984.