An Alternative Architecture for Objects: Lessons from the MUSHROOM project

Mario Wolczko* and Ifor Williams
Department of Computer Science
University of Manchester

September 12, 1993

Over the last six years, the MUSHROOM project at the University of Manchester has investigated alternative computer architectures for support of object-oriented systems. After much simulation and design study, we designed an architecture which we believe is especially suited to the characteristics of object-oriented systems, and constructed a prototype implementation of that architecture. Detailed quantitative evaluation of this architecture is the subject of ongoing work[1].

The MUSHROOM architecture follows in the tradition of RISC support for object-oriented languages pioneered by SOAR [5]. It possesses "classical" RISC features such as a load-store memory interface, delayed branches and register windows, but also includes some features specifically for the support of dynamic object-oriented languages, such as tagged memory and parallel tag-checking arithmetic and comparison instructions (similar to features in SOAR and the Symbolics Lisp Machines [3]). However, it goes much further than other RISC architectures in its provisions for object-based storage.

The load-store interface of the MUSHROOM instruction set does not see memory as a contiguous address space of bytes or words, but rather as a

*To whom correspondence should be addressed, at: Sun Microsystems Labs. Inc., 2550 Garcia Avenue, MS MTV29-116, Mountain View, CA 94043, U.S.A. email: Mario.Wolczko@eng.sun.com

[1]Using a simulator, rather than the prototype, for non-technical reasons too obscure to cover here

fine-grained segmented store, each segment containing a single object. A load or store instruction uses a two-part address to access a word in memory, consisting of the segment or *object identifier*, and an offset selecting a word within that object. The object identifier contains no indication of the physical location of the object. Because objects are typically small, we chose to limit offsets to eight bits, constructing larger objects in software from smaller objects, much as a filesystem presents the illusion of a file as a contiguous stream of bytes which is in reality implemented as separate disk blocks.

1 Benefits of segmented addressing

By abandoning the classic linear storage model of the von Neumann machine for a two-level addressing structure (based on location-independent object identifiers) we have been able to address many problems that have plagued the implementations of object-oriented languages on conventional architectures.

These include:

- *The design of cache and memory structures to provide fast access to objects.*

  The MUSHROOM cache structure uses the two-part addresses to quickly access a datum in cache. Each cache line associates directly on the two-part virtual address; no address translation is required. Accesses are pipelined, and the cache itself can respond to hits in such a way that no delay slot is required after an access.

  Simulations indicate that the effective access time is 1.8 cycles for reasonable cache sizes, including miss overheads [6].

- *Efficient management of caches and address translation.*

  Each cache line contains (a part of) only a single object, exploiting object locality to improve hit rates. Measurements indicate that temporal locality within an individual object is extremely high – for the Smalltalk programs we measured, there was a 99% likelihood than if an object was accessed (as data) it would be accessed again within the next 100 memory operations [8]. Sharing cache lines would lead to more complexity which slows cache operation.

When a cache miss occurs, address translation is required. This uses the cache hardware itself to provide a multi-level virtual object table, part of which can be cache-resident. The real address of each object is held in a field in an object table object (OTO); the real address of the OTO is in an object table table object (OTTO), and so on, up four levels, up to a root object which is always at a predetermined real address.

The identity of an object's OTO is a simple function of the object's identity. When a cache miss occurs, a software trap handler computes this function, and attempts to access the object's real address in its OTO; if this succeeds, the object is loaded into cache, if it fails, the procedure is applied recursively to the OTO to access the OTT, and so forth.

The basic idea of in-cache address translation is taken from SPUR [2].

- *Effective cooperation between the storage hierarchy management and garbage collector, avoiding conflicts between the requirements of the garbage collector and demands on virtual memory.*

  The cache structure is visible to software, and in particular to the garbage collector, which exploits the structure of the cache to maintain its invariants and has a portion of the cache dedicated to its own use to avoid conflicts with the mutating processes [9].

  Garbage collection is achieved by a multi-level generational scheme [4], with the data cache serving as the place where objects are created, and the primary area of garbage collection. A second level of garbage collection takes place in main memory, using an incremental on-the-fly collector [1]. Hence, the cache-memory boundary serves as a barrier (write barrier for the primary collector, and read barrier for the incremental collector), requiring maintenance only at cache flushes and fills and not for memory operations that hit the cache.

- *Efficient transfer of objects between RAM and secondary storage.*

  Location-independent addressing is exploited by relocating objects within the storage hierarchy to exploit temporal locality. This increases the effectiveness of transfers between main memory and disk.

Sometime after an objects becomes inactive, it is likely to be ejected from the cache. When this occurs, it is added to a list of all inactive objects in main memory (and will be removed from the list if accessed before being swapped to disk). When the virtual memory system determines that some objects must be expelled from memory, it takes as many objects as will fit on a disk block from the head of the inactive list (i.e., the least recently used objects in main memory), and ejects those. Thus, objects are grouped and regrouped according to the history of activity. Simulations based on traces taken from Smalltalk programs suggest that this technique could reduce paging traffic substantially, by increasing the useful content of a disk block fetched into memory [7].

- *Efficient object allocation, garbage collection and deallocation.*

  Because the cache is software-visible, when possible an object is allocated directly into free cache lines, with no redundant memory accesses to write back garbage or fetch unused data.

  The primary garbage collector, an in-cache scavenger, reclaims in-cache garbage without requiring any accesses to main memory, which enables an upper bound to be placed on garbage collection pauses directly related to the size of the data cache [9].

- *The provision of sufficient address space for large object-based systems.* The prototype implementation, using tagged 40-bit main memory words consisting of eight tag bits and 32 identifier bits, can directly access up to 2^{28} objects in virtual memory. Extension to larger word sizes and address spaces is straightforward.

2 Applications to conventional architectures

Despite the advantages of this architecture, it is clear that widespread adoption of radically new architectures is not going to happen in the forseeable future. The question then arises: How can these benefits be integrated into existing systems with minimal disruption?

We believe that the benefits of these techniques can be adopted in more conventional architectures (e.g., SPARC) with only a few changes to the memory system interface. The basic idea is to mimic the object-based addressing scheme as closely as possible. One way is to reserve virtual address space in a conventional system so that the two-parts of an object address can be concatenated to form a single conventional address. Using a maximum object size of 256 words, and assuming that the offset occupies the least significant bits of the address, this would mean that every object started on a 256-word virtual address boundary. As this would limit the number of objects to 1/256th of the size of the virtual address space, it is important that virtual address space be large enough to cope (e.g., on a 32-bit system we could accomodate 2^{24} objects, which is enough for small programs; fortunately 64-bit machines are becoming more widespread). Further, to avoid wasting the memory implied by this scheme, we would require a more sophisticated address translation mechanism than provided by current hardware.

Our proposal is in fact that no hardware be provided, or, more precisely, that it be possible to disable existing address translation hardware, instead using a software handler to deal with cache misses. This also implies that caches must associate on virtual addresses rather than real addresses, but this again implies less complexity (at perhaps the expense of more cache tag bits). Of course, it is essential that the expense of entering and exiting the handler be minimal, and the cache be large enough to deliver excellent hit rates, otherwise cache miss handling will dominate CPU time.

Once the cache is software-controllable, all the techniques used in the MUSHROOM memory system can be implemented (multi-level garbage collection, dynamically grouped virtual memory, in-cache allocation and deallocation, etc.) This facility could be made a software-controlled option so that existing non-object-based software, more suited to the existing memory structure, could run unaffected.

References

[1] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Comm. ACM*, 21(11):966–975, Nov. 1978.

[2] M. Hill et al. SPUR: A VLSI multiprocessor workstation. *IEEE Computer*, 19(11):8–22, Nov. 1986.

[3] A. R. Pleszkun and M. J. Thazhuthaveetil. The architecture of Lisp machines. *IEEE Computer*, 20(3):35–44, Mar. 1987.

[4] D. Ungar. Generation scavenging: A non-disruptive, high performance storage reclamation algorithm. In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, PA, May 1984. ACM SIGSOFT/SIGPLAN.

[5] D. M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* MIT Press, 1987.

[6] I. Williams and M. Wolczko. An object-based memory architecture. In A. Dearle, G. M. Shaw, and S. B. Zdonik, editors, *Implementing Persistent Object Bases: Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 114–130. Morgan Kaufmann Publishers, Inc., 1991.

[7] I. W. Williams, M. I. Wolczko, and T. P. Hopkins. Dynamic grouping in an object oriented virtual memory hierarchy. In J. Bézivin, J.-M. Hullot, P. Cointe , and H. Lieberman, editors, *Proceedings of the 1987 European Conference on Object-Oriented Programming, Lecture Notes in Computer Science*, volume 276, pages 79–88. Springer-Verlag, Paris, June 1987.

[8] I. W. Williams, M. I. Wolczko, and T. P. Hopkins. Realisation of a dynamically grouped object-oriented virtual memory hierarchy. In *Proceedings of the Workshop on Persistent Object Systems: Their Design, Implementation and Use*, pages 298–308, Aug. 1987. Persistent Programming Research Report, Universities of Glasgow and St. Andrews (PPRR-44-87).

[9] M. Wolczko and I. Williams. Multi-level garbage collection in a high-performance persistent object system. In *Persistent Object Systems, San Miniato, 1992*, pages 396–418. Springer-Verlag, San Miniato, Italy, 1993.