Using Virtual Addresses as Object References

Jeff Chase, Hank Levy, and Ashutosh Tiwary

Department of Computer Science and Engineering FR-35
University of Washington
Seattle, WA 98195
USA

Abstract

Most object-oriented systems that support distributed or persistent data structures use software-interpreted object identifiers for inter-object referencing. These identifiers are often called pointer *surrogates*, because they are used as substitutes for virtual memory pointers. Surrogates are the traditional means of referencing objects that are viewed as existing "outside" of an application's virtual address space, e.g., because they reside in a database.

An alternative to surrogates is to use ordinary virtual addresses for inter-object referencing. Usually (but not always) this involves mapping distributed or persistent data into specified parts of the application's address space, relying on page faults to trap and resolve references to nonresident data.

The choice between these two referencing schemes involves tradeoffs. Virtual addresses promise easier integration with programming languages and faster local pointer dereferences, whereas surrogates can simplify object-based aspects of storage management, such as garbage collection and heap compaction.

We believe that the key justifications for surrogate pointers are eliminated by 64-bit address spaces and proper operating system virtual memory support. When these facilities are available, the flexibility of surrogate pointers may not be worth their cost. We are investigating approaches to storage management that combine object-based and page-based techniques, and are compatible with the use of virtual addresses as object references.

1 Introduction

While most programming languages use virtual addresses as inter-object references (*pointers*), support for distributed or persistent objects is typically added through the use of *surrogates*. A surrogate is an object name that is interpreted by software on pointer dereference, typically by hashing or indexing into a table of *object descriptors* that point to the actual objects. Surrogate pointers have been used in distributed programming languages (e.g., Emerald [Jul et al. 88]), persistent object stores (e.g., Mneme [Moss 89]), object-support systems (e.g., SOS [Shapiro et al. 89] and Guide [Balter et al.]), and most commercial object-oriented database systems.

Surrogate pointers have two major drawbacks. First, they are more expensive to dereference than ordinary virtual addresses, which can affect application performance. Second, surrogates require tight integration with the language environment to preserve transparency, inhibiting support for

This work was supported in part by the National Science Foundation under Grants No. CCR-8619663 and CCR-8907666; by the Washington Technology Center; and by Digital Equipment Corporation through the Systems Research Center, DECwest Engineering, and the External Research Program.

off-the-shelf language implementations, particularly of *laissez-faire* programming languages such as C. While pointer swizzling is one approach to sidestepping these problems, we believe that swizzling is an incomplete solution that introduces many new problems.

Our position is that (1) the performance and simplicity of bare virtual addresses offers significant advantages over surrogate pointers, and (2) architectural and operating system advances will soon overtake traditional arguments for using surrogates. Specifically, we believe that 64-bit machines will become commonplace, eliminating the primary reason for surrogates, i.e., to name more objects than will fit in a traditional 32-bit virtual address space. In addition, extensible virtual memory interfaces have led to widespread use of memory mapping (which does not require surrogate pointers) as a basis for distribution and persistence.

As a result of these trends, we are building a new operating system to further simplify use of this technique. This system, called Opal [Chase et al. 92a], provides a single virtual name space that includes persistent storage and is shared by all applications. Opal encourages the use of virtual address pointers rather than surrogates because it assures uniform and stable virtual address bindings for all objects. However, uniform virtual addressing is not a panacea; problems such as garbage collection, heterogeneity, and efficient storage management continue to demand some help from language environments (i.e., the compiler and runtime system). This leads to a dilemma, because virtual address pointers conflict with traditional language-based approaches to managing distributed and persistent objects, which are often based on or intertwined with the use of surrogate pointers. In Section 3 we argue that this dilemma is less severe than it appears, and that many solutions to these storage management problems are compatible with the use of virtual address pointers.

2 Operating System Support for Shared Virtual Storage

Operating system advances, particularly innovative uses of virtual page protections [Appel & Li 91], are the catalyst for increased use of virtual address pointers. For example, extensible memory management interfaces (as in Mach and Chorus) have produced a flood of specialized paging servers that support persistent and distributed virtual storage transparently to the language implementation. The database community is increasingly interested in this approach; Bubba [Copeland et al. 90], Camelot [Eppinger 89], Cricket [Shekita & Zwilling 90], and ObjectStore [Lamb et al. 91] are recent examples of memory-mapped database systems.

Despite these advances, today's operating systems do not adequately support the use of virtual address pointers in shared and persistent data. Sharing requires that stored virtual addresses have a consistent *interpretation*; in most current systems, this requires that applications agree upon and reserve a fixed virtual address range where the data will be mapped. In general, sharing patterns must be known in advance and hardwired into the applications.

To resolve this problem, some systems (e.g., COOL-2 [Lea et al. 91]) arrange for each application to reserve a large chunk of its address space, which is then centrally allocated by the operating system to shared data segments. In the Opal operating system we extend this approach by coordinating *all* virtual address usage. The system assigns unique virtual address ranges to all data segments, including persistent data and all transient data in use by all applications. In effect, the system defines a *single virtual address space* shared by all applications. This does not entail sacrificing memory protection; applications execute in private protection domains, and segments are shared only by mutual agreement. Also, the system can accommodate multiple models of virtual storage within this single address space, through the use of external paging servers.

Single address space systems are controversial because they prevent applications from using the same virtual addresses to name different data. Our goal in this paper is not to justify this model; instead, we refer you to [Chase et al. 92b]. Briefly, our position is that 64-bit address spaces are large enough to make a single address space viable even across a local area network, and that in a 64-bit world there are no significant benefits to private virtual address spaces. A single address space is preferable because virtual addresses are *unambiguous* in such a system; this assures the validity of pointers in shared data, and simplifies the use of virtually addressed data caches.

By virtue of its global virtual address space, Opal assigns a unique virtual address to every object. In particular, applications can decide to share or store data objects after they been created, without allocating shared object names in advance, and without the cost of indirection. Thus, virtual addresses can be used for inter-object referencing in a fully general way, even for shared and persistent data. Of course, Opal does not *require* that inter-object references be virtual addresses; it does not impinge on the right of the application or language environment to choose to represent references in some other way, e.g., by using symbolic names, surrogates, or an extra level of virtual address indirection.

3 Managing Shared Virtual Addresses

Although Opal's global virtual address space provides the basic infrastructure for supporting shared and persistent objects, many aspects of object management (such as garbage collection and prevention of dangling references) must be relegated to the language implementation and its runtime system. Some language environments will support these aspects well (e.g., those based on a strongly-typed object-oriented language), and others will support them badly or not at all (e.g., C running above a recoverable virtual store). This is acceptable because safe environments can always be insulated from unsafe ones through the use of protection domains and accounting.

Unsafe language environments are easy to build, but they are of limited value. The important question is: given Opal's support for shared virtual storage via a single uniform address space, is it possible to build safe and fully functional language environments for persistence and distribution that use bare virtual addresses for object references? The major hurdle is that an object cannot be moved to a different part of the address space without updating all pointers to it. Below we describe four areas of object management for which this constraint could cause problems, and suggest that some solutions are compatible with virtual address pointers, for either of two reasons: (1) they do not require that objects be moved, or (2) objects can be moved without pointer indirection.

(1) **Garbage collection.** The type system of a strong language can supply enough information to locate pointers in memory, without relying on tagged surrogates. The runtime system can use this information to drive garbage collection. Mark-and-sweep garbage collection algorithms do not require that objects be moved. Some algorithms for concurrent copying garbage

collection can move objects without pointer indirection, by using virtual page protections to trap dereferences through the old pointers [Appel et al. 88].

(2) **Fragmentation.** Storage may become fragmented if the heap is not compacted; even if the system allows the heap manager to release physical pages out of the middle of the heap, a page cannot be freed until all objects overlapping it have been reclaimed. Ordinarily this is not a concern in non-persistent environments. We hope that proper heap management can reduce fragmentation to reasonable levels for most object allocation patterns, even in persistent environments. Alternatively, heap compaction can be achieved without indirection as a side effect of concurrent copying garbage collection, mentioned above.

(3) **Coherency and Updates.** Most systems today propagate updates to distributed and recoverable memory on a page granularity (e.g., write all modified pages on transaction commit). We prefer object-grained updates when the system knows how the data is partitioned into objects, and which objects have been modified. Again, strong type systems can supply this information. In this case, object-grain updates are compatible with virtual address pointers, although the difficulty of moving objects does constrain the implementation. For example, Amber [Chase et al. 89] uses such a hybrid approach for distributed memory, and Bubba for recoverable memory.

(4) **Heterogeneity.** Heterogeneity can be supported by translating objects (e.g., swapping bytes or translating floating point representations) and rebinding their methods as they move between machines. Again, this requires information from the type system, but it is independent of the use of surrogates. However, objects will grow or shrink if the source and target architectures have different size representations for primitive data types, which may force the system to move them to a different part of the address space. We assume that the network consists entirely of 64-bit machines; this is a reasonable working assumption for the long term.

4 Summary and Conclusion

In this paper, we have:

(1) asserted that ordinary virtual address pointers, rather than surrogates, should be used whenever possible for inter-object references in distributed and persistent objects;

(2) described virtual memory support to simplify the use of virtual address pointers on emerging 64-bit architectures;

(3) outlined several aspects of storage management that cannot be supported by the virtual memory system alone;

(4) argued that these aspects of storage management can be supported (as they are today) by language environments, particularly those based on strongly typed object-oriented languages, even if virtual address pointers are used.

In conclusion, we believe that hybrid approaches to distributed and persistent storage management, combining virtual memory and object-based techniques, are a promising and increasingly important research area in object-oriented systems.

References

[Appel & Li 91] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proc. of the ACM Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[Appel et al. 88] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent garbage collection on stock multiprocessors. *SIGPLAN Notices (Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation)*, 23(7), July 1988.

[Balter et al.] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. LeDot, H. N. Van, E. Paire, M. Riveill, C. Roisin, X. R. de Pina, R. Scioville, and G. Vandôme. Architecture and implementation of Guide, an object-oriented distributed system. *Computing Systems*, 4(1):31–68.

[Chase et al. 89] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 147–158, December 1989.

[Chase et al. 92a] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. Lightweight shared objects in a 64-bit operating system. Technical Report 92-03-09, University of Washington, Department of Computer Science and Engineering, March 1992.

[Chase et al. 92b] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. How to use a 64-bit virtual address space. Technical Report 92-03-02, University of Washington, Department of Computer Science and Engineering, February 1992.

[Copeland et al. 90] G. Copeland, M. Franklin, and G. Weikum. Uniform object management. In F. Bancilon, C. Thanos, and D. Tsichritzis, editors, *Advances in Database Technology – International Conference on Extending Database Technology 1990 (Lecture Notes in Computer Science 416)*, pages 253–268. Springer-Verlag, 1990.

[Eppinger 89] J. L. Eppinger. *Virtual Memory Management for Transaction Processing Systems.* PhD dissertation, Carnegie Mellon University, February 1989. CMU-CS-89-115.

[Jul et al. 88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Lamb et al. 91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10), October 1991.

[Lea et al. 91] R. Lea, P. Amaral, and C. Jacquemot. COOL-2: an object-oriented support platform built above the Chorus micro-kernel. In *Proceedings of the International Workshop on Object-Oriented Operating Systems*, October 1991.

[Moss 89] J. E. B. Moss. The Mneme persistent object store. COINS Technical Report 89-107, University of Massachusetts at Amherst, October 1989.

[Shapiro et al. 89] M. Shapiro, P. Gautron, and L. Mosseri. Persistence and migration for C++ objects. In *Proceedings of the Third European Conference on Object-Oriented Programming*, July 1989.

[Shekita & Zwilling 90] E. Shekita and M. Zwilling. Cricket: A mapped, persistent object store. In *Proceedings of the Fourth International Workshop on Persistent Object Systems: Design, Implementation and Use*, September 1990.