Opal: A Single Address Space System for 64-bit Architectures

Jeff Chase, Hank Levy, Miche Baker-Harvey, Ed Lazowska

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

The recent appearance of architectures with flat 64-bit virtual addressing opens an opportunity to reconsider the way our operating systems use virtual address spaces. We are building an operating system called Opal for these wide-address architectures. The key feature of Opal is a single global virtual address space that extends to data on long-term storage and across the network. In this paper we outline the case for the use of a single virtual address space, present the model of addressing and protection used in Opal, and discuss some of the problems and opportunities raised by our approach.

1 Introduction

The Opal project is an investigation into the effect of wide-address architectures on the structure of operating systems and applications. Our premise is that the next generation of workstations and servers will use processors with 64-bit data paths, and sparse, flat, 64-bit virtual addressing. The MIPS R4000 [MIP 91] and Digital's Alpha family [Dobberpuhl et al. 92] are recent examples of the trend to wider addresses. Our goal is to determine how software can best exploit the large virtual address spaces of these emerging architectures.

We view the move to 64-bit addressing as a qualitative shift that is far more significant than the move to 32-bit architectures in the 1970s. 64-bit architectures remove basic addressing limitations that have driven operating system design for three decades: consider that a full 64-bit address space consumed at the rate of 100 megabytes per second will last for 5000 years. On 32-bit architectures virtual addresses are a scarce resource that must be multiply allocated in order to supply executing programs with sufficient name space. Small address spaces are the reason for the traditional model of virtual storage that dominates today's operating systems, in which each program executes in a private virtual address space.

We are building a 64-bit operating system called Opal with a *single virtual address space* that maps all primary and secondary storage across a network [Chase et al. 92a, Chase et al. 92b]. This simply means that virtual address usage is coordinated so that any piece of shared data is named with a unique address by all programs that access it. Protection is independent of this global name space: an executing program can address any piece of data in the system, but programs execute within *protection domains* that restrict their access to memory. An Opal protection domain is in many ways the analog of a Unix process. For example, there is typically one domain per executing application, containing some private data, threads, and RPC connections to other domains. The difference is that an Opal domain is a private set of access privileges for globally addressable pages rather than a private virtual naming environment. Opal domains can grow, shrink, or overlap arbitrarily, by sharing or transferring page permissions.

No special hardware support for Opal is assumed or required. We believe that Opal could run efficiently on DEC Alpha and MIPS R4000 processors. However, single address space operating systems can benefit from hardware that is optimized for the way they use virtual memory [Koldinger et al. 92].

This work was supported in part by the National Science Foundation under Grants No. CCR-8619663, CCR-8907666, and MIP-9058-439; by the Washington Technology Center; and by Digital Equipment Corporation through the Systems Research Center, DECwest Engineering, the External Research Program, and the Graduate Engineering Education Program.

2 Why a Global Address Space?

A common virtual address space can eliminate obstacles to sharing and cooperation that are inherent in the traditional (e.g., Unix) model of processes excecuting in private virtual address spaces and communicating through messages or byte-stream files. The basic problem with these systems is that a stored virtual address (pointer) has no meaning beyond the boundaries of the process that stored it; information containing pointers is not easily shared because pointers may be interpreted differently by each process that uses the data.

A single address space system separates naming and protection so that virtual addresses have a globally unique interpretation. This simplifies sharing in two ways. First, addresses can be passed between domains in messages or shared data; any byte of data can be named with the same virtual address by any protection domain that has permission to access it. Second, a domain can import data containing embedded virtual addresses without risk of a name conflict with other data already in use by that domain. Uniform virtual addressing in single address space systems has some effects that are obvious and some that are more subtle. Here are the fundamental claimed benefits:

- *Shared memory is easier to use.* Linked data structures are meaningful in shared regions. Creative use of shared memory can reduce the need for explicit communication between programs executing in separate protection domains. This is important because the communication mechanisms popular today (e.g., RPC) are based on data copying and protection domain switches, which have increased in cost relative to integer performance on recent processors [Ousterhout 90, Anderson et al. 91].
- *Hardware-based memory protection is cheaper and more flexible.* Protection domains can be created more efficiently if the system is freed from setting up a new address space for each domain. Also, protection can be used when it is needed without introducing nonuniform naming of shared code or data; this allows complex programs to be decomposed into multiple domains that share some of their data structures. In Opal, it is possible to create a domain to execute any procedure call, passing it arbitrary data structures as its input and receiving arbitrary structures as its output.
- *Persistent storage can be integrated into the global virtual address space.* A single-level store allows programs to store data structures on disk without the expense and complexity of converting pointers to a separate format. Many systems today support mapped files, but pointers are preserved only if addresses assigned to the file are unused in every domain that references the file. Only a single address space system can guarantee this property in a fully general way.

The argument for shared memory may not sound convincing at first. Conventional wisdom says that "shared memory compromises isolation", but we claim that some uses of shared memory are inherently no less safe than more expensive mechanisms for cooperation. Untrusting domains can limit their interactions by restricting the scope of the shared region and the way that it is used. Memory can be shared sequentially by transferring access permissions, so that untrusting programs can verify data before using it. Also, asymmetric trust relationships are common and can be exploited. For example, a server domain never trusts its clients, but each client must trust the server to some degree; servers may pass data in memory that is read-only to clients, or even write directly into a client's private memory. Similarly, a child domain always trusts its parent, though the parent never fully trusts the child.

We believe that improved support for sharing and storage of data structures is useful for integrated software environments, which are difficult to build on private address space systems. For example, CAD systems are sometimes structured as groups of tools operating on a common data structure in various ways, e.g., editors, design verifiers, simulators, and so on.

3 The Opal System

In this section we present the Opal model of virtual storage and describe some of its benefits. We focus on Opal's use of protection and the handling of shared code and data, avoiding other aspects of the system, such as execution structures, storage management, and cross-domain control transfers (RPC). The basic model is threads executing within protection domains and making synchronous RPC calls to other domains. All threads executing within a domain have the same protection identity. Any thread can create a new protection domain, attach code and data to it, and start threads in it. Threads and synchronization primitives are implemented in a standard runtime library; the user-level approach is well-suited to handling synchronization in shared and persistent data structures.

Opal is designed for a distributed environment composed of one or more nodes connected by a network. Each node contains physical memory, one or more processors attached to that memory, and possibly some long-term storage ("disk"). We assume the processors are homogeneous. The global address space is extended across the network by placing servers on each node that maintain a partitioning of the address space, both to ensure global uniqueness and to allow data to be located from its virtual address. The problems are similar to those faced by other systems for distributed name management.

3.1 Virtual Segments and Addressing

To simplify access control and virtual address assignment, the global address space is partitioned into *virtual segments*, each composed of a variable number of contiguous virtual pages that contain related data. Each segment occupies a fixed range of virtual addresses, assigned when the segment is created, and disjoint from the address ranges occupied by all other segments.

Before a protection domain can complete a memory reference to a segment it must establish access to the segment with an explicit *attach* operation. Domains attached to a given segment always share a physical copy of its data if they are on the same node. All memory references use a fully qualified flat virtual address, so any domain with sufficient privilege can operate on linked data structures in any segment, without name conflicts or pointer translation. Pointer structures can also span segments, so different access controls can be placed on different parts of a connected data structure.

A domain must have special permission to attach a segment. Segments are named by capabilities (based on capabilities in the Amoeba system [Mullender & Tanenbaum 86]) that can be passed between protection domains through shared or persistent memory. A segment capability confers permission to its holder to attach the segment and address its contents directly with **load** and/or **store** instructions. A memory reference to an unattached segment is reflected back to the domain as a *segment fault* to be handled by a standard runtime package. Some segments can be attached dynamically in response to segment faults; this is useful if the application is navigating a pointer graph spanning multiple segments. Segments are attached eagerly when the application can anticipate what storage resources it needs. In this case segment faults are used to trap and report stray memory references.

3.2 Persistent Memory

A *persistent* segment continues to exist even when it is not attached to any domain. A *recoverable* segment is a persistent segment that is backed on nonvolatile storage and can survive system restarts. All Opal segments are potentially persistent and recoverable. Persistent virtual memory can be viewed as a replacement for a traditional filesystem; recoverable segments share many of the characteristics of mapped files. The policies for managing persistence and recovery vary according to segment type, and are beyond the scope of this paper (our prototype mechanisms are quite primitive).

Since persistent segments are assigned virtual address ranges in the same way as other segments, Opal supports a true *single-level store* with uniform addressing of data in long-term storage. Access to persistent data is transparent, though explicit operations may be used to commit or flush modified data in segments with strong failure recovery semantics. A domain can name a segment by the address of any piece of data that lies within it, even if the segment is not yet attached. In essence, this allows pointers between files; a file can be mapped without specifying a symbolic name, simply by dereferencing a pointer. A symbolic name space for data exists above the shared virtual address space, in the form of a name server that associates symbolic names with segment capabilities or arbitrary pointers.

3.3 Code Modules

Executable code resides in persistent segments called *modules* that contain a group of related procedures. A module is *pure* if it makes no static assumptions about the load address of any data that it operates on. Linking utilities make modules pure by expressing all private static data references as offsets from a base register. Pure modules are statically linked into the global address space at their assigned address. There are two distinct benefits from global linking of code modules.

First, domains can dynamically load (attach) modules without risk of name conflicts in the code references, and without the overhead of linking at runtime. There is no need to know at domain creation time what code will run in the domain; any domain can call any procedure it has access to simply by knowing its address, even if the code was compiled after the domain was activated. For example, a domain can call through procedure pointers passed to it in shared data. Dynamic loading also allows a parent to choose whether or not to create a new protection domain for a child; it is no longer necessary to create new protection domains for trusted programs simply because they are statically linked to assume a private name space.

The second benefit of global linking is that pure modules can be freely shared between domains. Comparable support for shared libraries in private address space systems requires dynamic linking and/or indirect addressing through linkage tables.

3.4 Reclamation

With an addressing model that permits and encourages sharing, the system must cope with the difficulty of managing that sharing. For example, it is not always clear when data can be deleted in a single address space system.

We believe that existing approaches to reclamation are applicable. Opal does not track references, it merely provides hooks to allow servers and runtime packages that manage shared segments to plug in their own reclamation policies. Our prototype places active and passive reference counts on each segment and provides a means for clients to manipulate those reference counts in a protected way. It is the client's responsibility (the runtime package and/or the language implementation) to use reference counts appropriately. If the clients do nothing then the resulting policy is exactly that of most process-based systems: a segment is deleted iff it is unattached and there are no symbolic names for it in the name server. Of course, clients may misuse the counts to prevent storage from *ever* being reclaimed. This is an accounting problem.

This approach reflects our view that reclamation should continue to be based on language-level knowledge of pointer structures, application-level knowledge of usage patterns, and deletion of data by explicit user command. The system only promises that domains cannot harm each other with reclamation errors unless they are mutually trusting. This does not mean that we are punting the issue of reclamation: we cling to the belief that support for simple but useful sharing patterns can be built relatively easily.

4 Why a Single Mapping?

One alternative to a global virtual address space is to reserve some regions of private address spaces for mapping shared data. For example, virtual addresses can be saved directly in a mapped file if the processes that use the file agree to always map it at the same address. Dynamic sharing patterns can be supported if the *system* (rather than the applications) coordinates address bindings.

This solution buys some of the benefits of a single address space without sacrificing private address spaces. In fact, our original proposal for global addressing on 64-bit architectures suggested such a hybrid approach [Chase & Levy 91]. We have now abandoned the hybrid approach for two reasons: (1) the mix of shared and private regions introduces dangerous ambiguity, and (2) the virtual memory hardware and software must continue to support multiple sets of address translations.

The pure single address space approach forces us to confront the the effects of eliminating private address spaces altogether. Here are some of the interesting issues: (1) handling full global allocation of address space, (2) linking in a shared address space, e.g., how to handle private static data references in shared code, (3) implications of discarding the Unix **fork** primitive, which assumes private address spaces, and (4) effect on existing programs of losing the contiguous address space abstraction.

5 Related Work

Cedar [Swinehart et al. 86] and its predecessor Pilot [Redell et al. 80] used a single virtual address space on hardware that supported only one protection domain. These systems had no hardware-based memory protection; both relied solely on defensive protection from the name scoping and type rules of a programming language. Our proposal generalizes this model to multiple protection domains.

The term "uniform addressing" was introduced in Psyche [Scott et al. 90], which also has a single virtual address space shared by multiple protection domains. Psyche uses cooperating protection domains primarily as a means of separating different components of a single parallel application with different models of parallelism and different scheduling needs. We are interested in more general sharing relationships, leading to some differences in our system abstractions. More importantly, we extend uniform addressing to encompass data on long-term storage and across multiple autonomous nodes in a network. However, most of the benefits claimed for context-independent addressing in Opal apply equally to non-persistent storage in Psyche.

Segmented systems (e.g., Multics [Daley & Dennis 68]) support uniform sharing to some degree. The first phase of address translation on segmented architectures concatenates a global segment identifier with a segment offset, yielding a *long-form* address from a global virtual address space. The segment identifier is retrieved from one of a vector of *segment registers* associated with the current domain. Domains define a local view of portions of the global address space by overlaying global segments into their private segment registers. Uniform addressing in these systems is subject to some or all of the following restrictions: (1) cross-segment pointers are not supported, (2) multiple pointer forms must be treated differently by applications, and (3) software must coordinate segment register usage to create an illusion of a single address space. The Hewlett-Packard Precision [Lee 89] differs from other segmented architectures in that it allows applications to specify long-form virtual addresses directly. However, long-form pointer dereference is expensive, so most software uses segmented addressing.

Most capability-based architectures [Organick 83, Levy 84] support uniform sharing of data structures. However, it is now possible to achieve these benefits using conventional page-based hardware, without the problems common to capability-based systems: (1) restrictive object-oriented data model, (2) noncompetitive performance, and (3) lack of support for distribution.

6 Status

We are currently prototyping Opal above an unmodified Mach 3.0 kernel on 32-bit MIPS R3000 machines. Our prototype consists of (1) standard runtime libraries (e.g., for threads, segment fault handler, etc.), (2) a Mach server that implements the Opal "kernel" abstractions, and (3) a ragged crew of linking and bootstrap utilities. Opal protection domains and segments are mapped by the server onto the Mach *task* and *memory object* abstractions. We will use the prototype environment to explore ways that the shared address space can improve the structure and performance of applications and the operating system. Our long term goal is to implement a single address space kernel to explore the effect of a single address space on operating system kernel mechanisms.

References

[Anderson et al. 91] Anderson, T. E., Levy, H. M., Bershad, B. N., and Lazowska, E. D. The interaction of architecture and operating system design. In *Proceedings of the Fourth Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–121, April 1991.

[Chase & Levy 91] Chase, J. S. and Levy, H. M. Supporting cooperation on wide-address computers. Department of Computer Science and Engineering Technical Report 91-03-10, University of Washington, March 1991.

[Chase et al. 92a] Chase, J. S., Levy, H. M., Baker-Harvey, M., and Lazowska, E. D. 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.

[Chase et al. 92b] Chase, J. S., Levy, H. M., Baker-Harvey, M., and Lazowska, E. D. 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.

[Daley & Dennis 68] Daley, R. C. and Dennis, J. B. Virtual memory, processes, and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[Dobberpuhl et al. 92] Dobberpuhl, D., Witek, R., Allmon, R., Anglin, R., Bertucci, D., Britton, S., Chao, L., Conrad, R., Dever, D., Gieseke, B., Hassoun, S., Hoeppner, G., Kowaleski, J., Kuchler, K., Ladd, M., Leary, M., Madden, L., McLellan, E., Meyer, D., Montanaro, J., Priore, D., Rajagopalan, V., Samudrala, S., and Santhanam, S. A 200mhz 64 bit dual issue CMOS microprocessor. In *International Solid-State Circuits Conference 1992*, February 1992.

[Koldinger et al. 92] Koldinger, E. J., Chase, J. S., and Eggers, S. J. Architectural support for single address space operating systems. Technical Report 92-03-10, University of Washington, Department of Computer Science and Engineering, March 1992.

[Lee 89] Lee, R. B. Precision architecture. *IEEE Computer*, pages 78–91, January 1989.

[Levy 84] Levy, H. M. *Capability-Based Computer Systems.* Digital Press, Bedford, Massachusetts, 1984.

[MIP 91] MIPS Computer Systems, Inc., Sunnyvale, CA. *MIPS R4000 Microprocessor User's Manual*, first edition, 1991.

[Mullender & Tanenbaum 86] Mullender, S. and Tanenbaum, A. The design of a capability-based operating system. *The Computer Journal*, 29(4):289–299, 1986.

[Organick 83] Organick, E. I. *A Programmer's View of the Intel 432 System.* McGraw-Hill, 1983.

[Ousterhout 90] Ousterhout, J. K. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[Redell et al. 80] Redell, D., Dalal, Y., Horsley, T., Lauer, H., Lynch, W., McJones, P., Murray, H., and Purcell, S. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[Scott et al. 90] Scott, M. L., LeBlanc, T. J., and Marsh, B. D. Multi-model parallel programming in Psyche. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 70–78, March 1990.

[Swinehart et al. 86] Swinehart, D., Zellweger, P., Beach, R., and Hagmann, R. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 4(8), October 1986.